Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Diseño de Red de Monitorización IoT con Microcontrolador WiPy.

Autor: Christian De Robles

Tutor: Manuel Ángel Perales Esteve

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales.

Diseño de Red de Monitorización IoT con Microcontrolador WiPy.

Autor:
Christian De Robles Gómez

Tutor:
Manuel Ángel Perales Esteve

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo Fin de Grado: Diseño de Red de Monitorización IoT con Microcontrolador WiPy.

Autor: Christian De Robles Gómez
Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

A modo de introducción del proyecto a presentar se expondrán unas lineas aclaratorias iniciales sobre el objeto y objetivo de la investigación que se propone así como las mejores que proporciona frente a los semejantes ya existentes.

En primer lugar exponemos el objetivo de diseño. Después de un simple análisis superficial de la realidad que nos rodea, desde un nivel cotidiano, laboral o incluso industrial, llegamos rápidamente a una primera evidencia, la cual no es otra que la necesidad de la monitorización de diferentes mediciones que nos proporcionan la gran variedad de sensores que contamos a nuestro alcance. Una de las medidas mas interesantes e importantes, la cual solíamos pasar por alto a no ser que nos encontráramos en atmósferas de trabajo muy concretas, y es la concentración del gas CO_2 en el ambiente. Dicha medición puede ser crucial para nuestra salud en lugares como nuestro ambiente de trabajo habitual o incluso un restaurante. Viendo la importancia de controlar los niveles de concentración de dicho gas en el ambiente, lo que se pretende diseñar en el siguiente proyecto es la posibilidad de desplegar una red de pequeños sensores de CO_2 en diferentes estancias, los cuales centralizarán los datos que se lean en un microcontrolador mediante el uso de la tecnología Bluetooth 4.0, a partir de ahora especificada como BLE. Dicho microcontrolador nos daría diferentes opciones de centralizar y observar las mediciones de los sensores que estén configurados para comunicarse con él.

Las funcionalidades definidas para esta idea que hemos comentado anteriormente, serían como hemos dicho la conexión BLE a distintos sensores. Para ello habría que configurar inicialmente todos y cada unos de los productores de datos en el microcontrolador central, y este iría recibiendo los datos de los sensores y los iría publicando según su modo de conexión que se explicarán mas adelante en profundidad. Se exige comprobar que todos los sensores están, vivos de algún modo y conectados, pues si no lo estuvieran habría que intentar volver a conectarse.

Así como los sensores siempre irán gobernados por un microcontrolador que lea su valor y lo envíe a través de un módulo BLE, esto satisface un requisito y es que se podría extender fácilmente a cualquier tipo de medida simplemente cambiando ese pack de $Sensor \rightarrow Microcontrolador \rightarrow BLE$, dicho conjunto lo llamaremos a

partir de ahora como ya lo hemos mencionado anteriormente, productor.

El microcontrolador que actúe como centralizador de dichos datos podrá tener diferentes modos de conexión, haciendo o no uso de una red WiFi. Esto nos da la capacidad de implementar nuestra red de sensores en lugares con conexión inalámbrica a internet, o incluso en lugares donde no llegue dicha conexión pues el mismo microcontrolador deberá actuar como servidor donde podemos ver dichos datos a través de la página que nos sirva el mismo.

Para tener un seguimiento de los datos que estamos recibiendo, a parte de ser estos datos visualizados, serán guardados en una tarjeta de memoria del tipo Micro SD, conectada al receptor de dichos datos, obteniendo siempre una copia local de nuestras mediciones independientemente del modo de funcionamiento.

Al implementar todas estos requisitos de funcionalidades que tenemos entre manos obtendremos la red global de monitorización de sensores a través de una red Bluetooth 4.0 de baja energía, BLE. Esto conlleva notables ventajas respecto al envío y recepción de mensajes haciendo uso de una red WiFi, pues dependeríamos siempre de la conexión a la misma e incluso del propio operador,así como sobrecargaríamos las ya saturadas redes empresariales e institucionales con conexiones parásitas que nos podremos ahorrar, haciendo uso de la radiofrecuencia.

Podemos observar que se trata de un paso mas en las redes de sensores que se implementan con Bluetooth Classic haciendo uso del bien conocido módulo HC-05. Respecto a dicha red obtenemos mejoras muy evidentes, las cuales se trataran en profundidad en apartados posteriores, como la calidad del modo de conexión entre dispositivos, el rango de conexión y el consumo de energía necesaria para la transmisión de datos, aunque dicha ventaja no haya sido un objetivo principal de nuestro proyecto.

Capítulo 2

Estado del arte

En este capitulo se explicará en detalle los diferentes tipos de tecnologías que se han usado a lo largo del desarrollo del proyecto

2.1. Tecnología Bluetooth

En primer lugar contamos con el uso de la tecnología Bluetooth, la mas importante de las que veremos en este capítulo pues en ella nos basamos para la comunicación de toda la red de sensores que se distribuyen.

Figura 2.1: Logo Bluetooth

Bluetooth no se trata en si de una tecnología nueva ni mucho menos, se trata de una especificación industrial para redes inalámbricas de uso personal, las conocidad como WPAN. Esta hace uso de la red de radiofrecuencia en la banda de los 2.4 GHz, al igual que muchos tipos diferentes de tecnologías de las que hacemos uso a diario,como las redes WiFi.Fue desarrolada por la compañía Bluetooth Special Interest Group, siendo esta una asociación de empresas sin ánimo de lucro que adoptaron Bluetooth como dicho estándar en el año 1989. El principal objetivo de desarrollo de Bluetooth fue mejorar las conexiones entre dispositivos para poder así eliminar cables, así como facilitando la sincronización de estos.

Los dispositivos que hacen uso del protocolo de comunicaciones Bluetooth se pueden clasificar en distintas clases, donde según la potencia de transmisión de estos obtenemos distintos rangos de alcance como observamos en la siguiente tabla.

Clase	Potencia Máxima Permitida (mW)	Alcance (m)
Clase 1	100	100
Clase 2	2.5	5-10
Clase 3	1	1
Clase 4	0.5	0.5

Cuadro 2.1: Clasificación Dispositivos Bluetooth Según Potencia

A lo largo de los años se han ido desarrollando diferentes especificaciones del mismo protocolo que han ido mejorando las capacidades de este mismo. En la siguiente tabla comparativa vamos a poder comparar como ha ido avanzando el ancho de banda a través de estas versiones, lo cual significa una mayor velocidad en la transmisión de datos.

Versión	Ancho de banda (Mbit/s)
1.2	1
2.0 + EDR	3
3.0 + HS	24
4.0	32
5.0	50

Cuadro 2.2: Clasificación Versiones Bluetooth Según Ancho De Banda

Las distintas especificaciones Bluetooth no van a ser descritas en profundidad, aunque si nos detendremos más en la especificación 4.0 de la cual hacemos uso en el proyecto, pero vamos a revisar de manera ligera la especificación 2.0 la cual es hoy día ampliamente utilizada en multitud de proyectos de bajo coste haciendo uso del módulo HC-05.

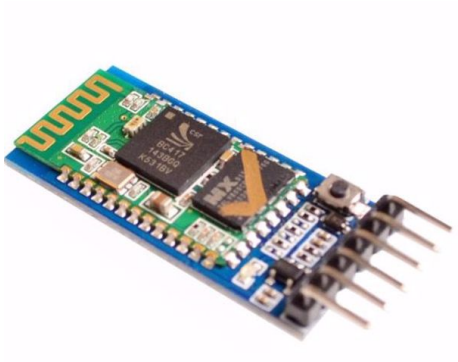

Figura 2.2: Módulo HC-05

El módulo HC-05 es una placa, conocida como "Módulo Bluetooth a Puerto Serie", lo que nos facilita enormemente la transmisión de datos entre dispositivos

haciendo uso del puerto serie de un microcontrolador. Este módulo es vendido por multitud de fabricantes con distintas variantes que principalmente hace uso del chip BC417 de Cambridge Silicon. Dicho modulo trabaja bajo la especificación Bluetooth 2.0, perteneciendo a la clase 2 de dispositivos descrita anteriormente y consumiendo alrededor de 30 a 50 mA. Para configurar su funcionamiento se hace uso de un listado de comandos AT que son recibido mediante puerto serie por el modulo. Mediante dichos comando podemos cambiar distintas opciones como el nombre que aparecerá visible para ser emparejado o su rol como maestro o esclavo.

Para comprender como funcionan las redes de dispositivos Bluetooth en primer lugar hay que saber que dicho dispositivo puede actuar como maestro o esclavo.

- **Maestro** Puede conectarse o permitir la conexión a 7 diferentes dispositivos esclavos, recibiendo y solicitando información de estos.
- **Esclavo** Solo tiene la capacidad de conectarse a un dispositivo maestro.

Con estos conceptos descritos podemos establecer un pequeño diagrama de conexión entre dos microcontroladores de manera inalámbrica. Uno de ellos realizará la lectura de cualquier tipo de dato que teniendo al modulo Bluetooth, como el HC-05 expuesto anteriormente, establecido como master, el cual le pasará la información a otro módulo configurado como esclavo.

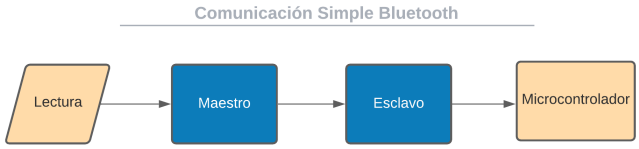

Figura 2.3: Diagrama de Conexión Básica Bluetooth

Como observamos en el diagrama anterior dicha estructura puede ser escalable a múltiples esclavos como comentábamos en su definición, donde la cual es bastante funcional si tenemos un dispositivo maestro como un teléfono móvil que envía música a diferentes auriculares.

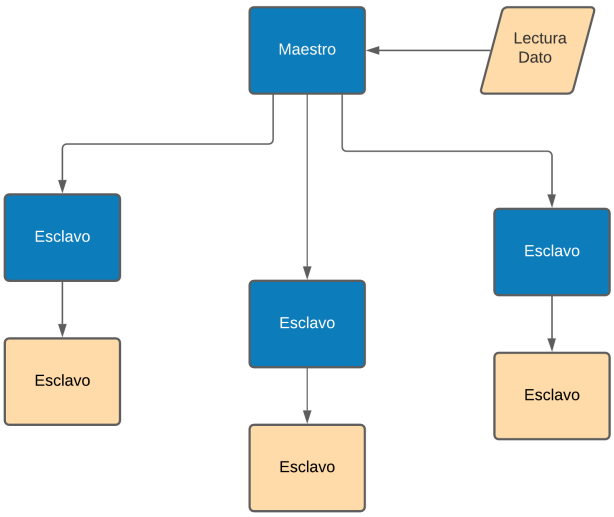

Figura 2.4: Diagrama Multi-Esclavo Bluetooth

Pero si queremos crear una red de sensores centralizada donde multitud de sensores envíen datos a un servidor vemos que no es posible pues tendríamos una red multi-maestro la cual por definición no es posible.

Ante la imposibilidad de implementar la funcionalidad requerida para el proyecto,

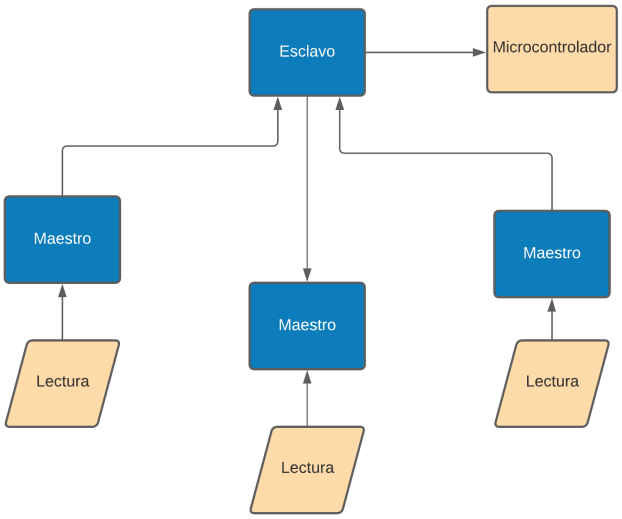

Figura 2.5: Diagrama Multi-Maestro Bluetooth

resolveremos dicho problema utilizando la especificación Bluetooth 4.0. Esta es el tejido esencial de nuestra red de sensores que hemos desplegado y la cual sera implementada resultado de sustituir el módulo HC-05 clásico que ya hemos comentado por otro que soporte dicha norma como el HM-10.

2.2. Especificación Bluetooth 4.0

En esta sección vamos a entrar más en profundidad en que es la especificación 4.0 de Bluetooth, como funciona y como la vamos a utilizar en el proyecto.
Cuando nos referimos a Bluetooth 4.0 nos referimos a un compendio de tecnologías

Figura 2.6: Logo de Bluetooth Smart

entre las que se encuentran el Bluetooth clásico que hemos visto anteriormente que consta de los protocolos existentes anteriormente., a este se añade el Bluetooth de alta velocidad, el cual se basa en WiFi para mejorar la tasa de intercambio. Por ultimo nos encontramos con el Bluetooth de baja energía, conocido como BLE (*Bluetooth Low Energy)* este subconjunto integra un protocolo completamente distinto para conseguir desarrollar enlaces mucho mas rápidos entre dispositivos, estando dirigidos a sistemas de muy bajo consumo, los cuales podriamos alimentar incluso con una pila de botón. Los chips que hacen uso de BLE están pensados para tener dos modos de uso, modo único donde solo integra BLE, y modo dual, integrando tanto Bluetooth clásico como BLE. Comercialmente esto pasaría a denominarse como la marca "Bluetooth SMART", teniendo dos tipos como hemos comentado.

- **Bluetooth SMART Ready.** Se refiere a los dispositivos en modo dual, normalmente un dispositivo, conocido como inteligente como podría ser un ordenador personal o un teléfono inteligente., los cuales pueden operar con periféricos *Bluetooth Classic* como *BLE*.
- **Bluetooth SMART.** Describe a un dispositivo que funciona en modo único, normalmente sensores con batería que solamente integran el protocolo de bajo consumo, requiriendo de otro dispositivo Bluetooth SMART Ready para funcionar..

2.3. Protocolos de Conexión en BLE.

Dentro de la especificación BLE que hemos descrito anteriormente, para poder comunicarnos con dispositivos que hacen uso de la misma, se describen protocolos internos de comunicación como los que serán descritos a continuación.

2.3.1. GAP

Para definir el marco en el que se permite la disponibilidad del dispositivo, la seguridad, la conectividad, así como todo lo relacionado con la estructura de la red que implementa este mismo, hacemos uso del *Generic Access Profile*, el perfil de acceso genérico el cual describiremos a partir de ahora como GAP. Este permite que nuestro dispositivo pueda adoptar do pares de roles dependiendo del uso que se le quiera dar.
El primer tipo de comunicación se establece con el par de roles Emisor y Observador, estos roles no están orientado a la conexión entre unos y otros si no al uso de la capacidad de mandar pequeños paquetes de datos que tiene un dispositivo BLE para ser visible a los demás, esto es conocido como *Advertising*.

- **Emisor.** Se trata de un rol activo donde el dispositivo se publicitaría a los demás.No permite la conexión con otros dispositivos y es comúnmente usado por los IBeacons, dispositivos que indican distancia mediante el envío de los paquetes de datos conocidos como *Advertising Data*.
- **Observador.** Se trata el complementario del anterior pues se trataría del rol pasivo, que adoptan dispositivos del tipo escáner que busca los datos que va publicando el emisor. Este rol tampoco inicia conexión pues solo esta preparado para recibir.

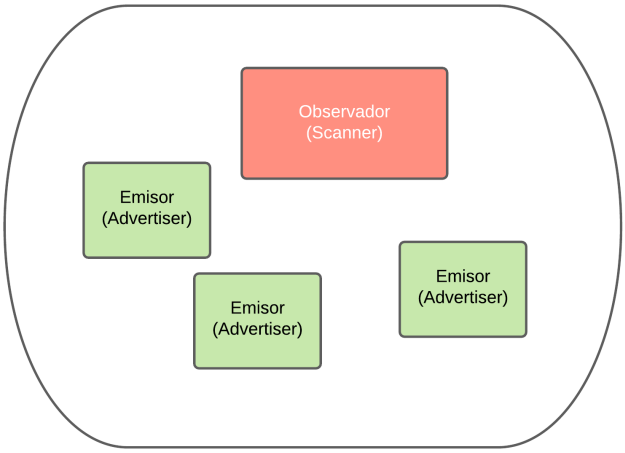

Figura 2.7: Red BLE con Roles Emisor/Observador

El siguiente tipo de comunicación esta orientada a la conexión mediante los roles Servidor y Periférico. Con esta red de comunicación tan parecida a lo lo visto anteriormente con Bluetooth clásico, es la que vamos a adoptar en la red de nuestro proyecto. Antes de establecer ninguna conexión funcionan igual que los roles anteriores, al establecer la conexión comparten datos de forma privada mediante comunicación GATT.

- **Servidor.** Dispositivo que inicia la conexión, se dedica a buscar datos publicados por los periféricos como *Advertising Data*, permitiendo la conexión a

múltiples dispositivos de este tipo. Este rol lo podemos asociar fácilmente a cuando desde la configuración de nuestro smartphone buscamos dispositivos Bluetooth a nuestro alrededor.

- **Periférico.** Mediante este rol no se inicia la conexión si no que se publicita la información que permite que el Servidor inicie la conexión. Solo permite estar conectado a un solo servidor.

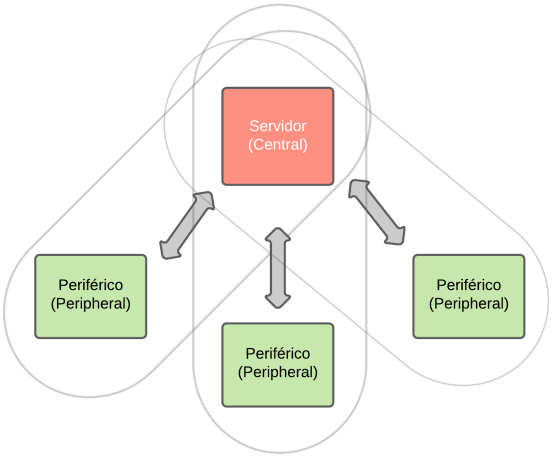

Figura 2.8: Red BLE con Roles Servidor/Periférico.

2.3.2. GATT

Como ya hemos comentado anteriormente mediante GAP explicamos la manera en la que se conectan diferentes dispositivos BLE unos con otros, pero en este caso GATT nos define la manera en que estos se comunican, es decir, la manera en que dos dispositivos pueden enviarse datos a través de Servicios y Características. En dicha comunicación se usa un protocolo conocido como ATT, usado para almacenar los servicios, características y sus datos en una tabla que a su vez usa identificadores de 16-bit en cada una de las entradas de la tabla.

Hemos de tener en cuenta que las conexiones son exclusivas, en el rol de Servidor/Periférico, cada periférico solo puede estar conectado a un dispositivo de rol central, como sería un teléfono móvil. Cuando dicho periférico se conecta al servidor, este dejará de anunciarse y los demás dispositivos no podrán detectarlo hasta que esta conexión acabe. Dicho esto entramos a fondo en la comunicación bidireccional que ocurre cuando estos dispositivos están en modo conectados. Para entenderlo esto debemos dejar claro como se relaciona la comunicación Servidor/Cliente en GATT.

- **Servidor.** Así es como se conecta el periférico que hemos descrito anteriormente, el cual contiene los datos de búsqueda ATT, así como las definiciones de los servicios y características del mismo.

- **Cliente.** Equiparable a lo que antes llamábamos servidor en GAP, pues el cliente es el que se encarga de enviar las solicitudes al servidor.

Según hemos descrito, la relación en nuestro proyecto sería la del modulo que incluye el sensor actuando como un servidor GATT y por otra parte, el microcontrolador Wipy actuando como cliente GATT.
Como ya hemos comentado las transacciones en este protocolo de comunicación se ejecutan mediante objetos anidados, que son los perfiles, servicios y características. El primer aglutinador se trata del perfil, este viene predefinido en una colección de

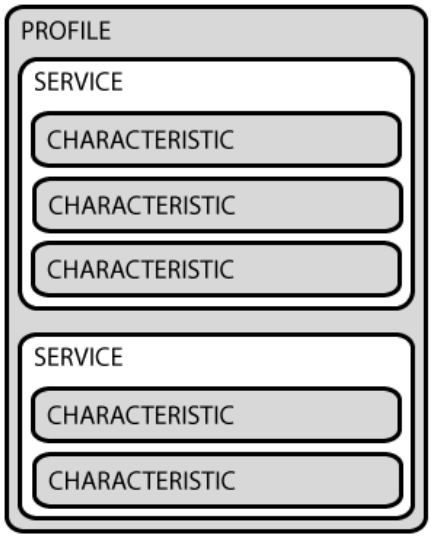

Figura 2.9: Estructura de Servidor GATT.

servicios predefinida por la especificación del fabricante o por la propia compañía desarrolladora de Bluetooth.En su web podemos consultarlo, pero el ejemplo más frecuente que se suele utilizar es el perfil de un dispositivo que se dedicará a monitorizar la frecuencia cardiaca,como un reloj inteligente. Este perfil incluiría el servicio de lectura de la frecuencia así como el servicio que utilizaría para comunicarse con otros dispositivos.

Un servicio se utiliza para separar los datos en diferentes entidades y están distinguidos por un identificador único de 16 bits. Los diferentes servicios adoptados oficialmente por BLE también pueden ser consultados en su web. Estos servicios incluyen datos específicos dentro de ellos llamados características, como veremos a continuación. Si seguimos con el ejemplo anterior del perfil de frecuencia cardiaca podemos ver que según la especificación oficial, este servicio esta identificado por el UUID 0x180D, el cual contiene 3 tipos diferentes de características en su interior, aunque solo es obligatoria la característica de la medición de la frecuencia cardiaca.

Finalmente en la atomización de la comunicación nos encontramos con las características dentro de los servicios, las cuales contiene un solo tipo de dato como por ejemplo en este caso el numero de pulsaciones por minuto que tiene el usuario.Al igual que el servicio estas son identificadas por un UUID de 16 bits predefinido,

siendo en este ejemplo que tenemos el 0x2A37. Las características son el elemento principal de la comunicación pues, es en ellas, donde podemos leer o escribir un dato que luego será leído por otro dispositivos.

2.3.3. Implementación BLE

Una vez conocemos como funciona el protocolo de comunicación GATT ya podemos entender, como nos vamos a comunicar en nuestro proyecto, donde en nuestros servidores vamos a implementar el módulo BLE HM-10. Este módulo esta fabricado

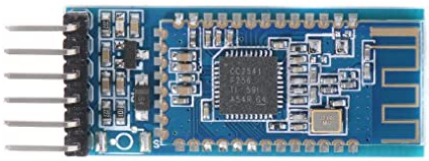

Figura 2.10: Módulo BLE HM-10.

por la empresa china Jinan Huamao technology Co. el cual no es mas que un BLE a puerto serie al igual que el HC-05 que describimos anteriormente pero haciendo uso de los chips CC254x de Texas Instruments, con el cual podemos implementar el protocolo de comunicación GATT. Según encontramos en la hoja de datos del módulo, estas son sus características técnicas.

Carcteristicas	Descripción
Versión Bluetooth	Especificación Bluetooth 4.0 BLE
Frecuencia	Banda de los 2.4 GHz
Alimentación	2.5V-3.3V
UUID Servicio	0xFFE0 (Modificable)
UUID Característica 1	0xFFE1 (Modificable)
UUID Característica 2	0xFFE2 (Modificable)
Consumo	(Activo) 8.5mA (Modo Bajo Consumo) 50uA-200uA

Cuadro 2.3: Características Técnicas HM-10.

Como vemos en sus datos técnicos este dispositivos incluye un perfil con un único servicio con dos características, de las cuales solo nos va a interesar una pues solo vamos a transmitir un valor, el valor que mida el sensor que nos atañe. Como podemos observar su consumo es extremadamente bajo en modo durmiente, comparado con el consumo del HC-05, obtenemos un consumo 3.5 veces menor en este en modo

activo no incluyendo el anterior un modo de bajo consumo. Gracias a esto podemos conseguir comunicación de datos con dispositivos a muy lejana distancia con una capacidad de sostenibilidad en cuanto a energía sorprendente.

Este dispositivo es configurado mediante los mismos comandos AT que hemos comentado anteriormente, solo tenemos que conectarlo a un puerto serie como sería el mismo del microcontrolador Arduino que vamos a usar, y enviarle AT+XXXX, pudiendo modificar su nombre, el id del servicio, el modo de conexión, etc. A continuación vamos a mostrar una captura de los comandos que nos ofrece si usamos el comando de ayuda AT+HELP.

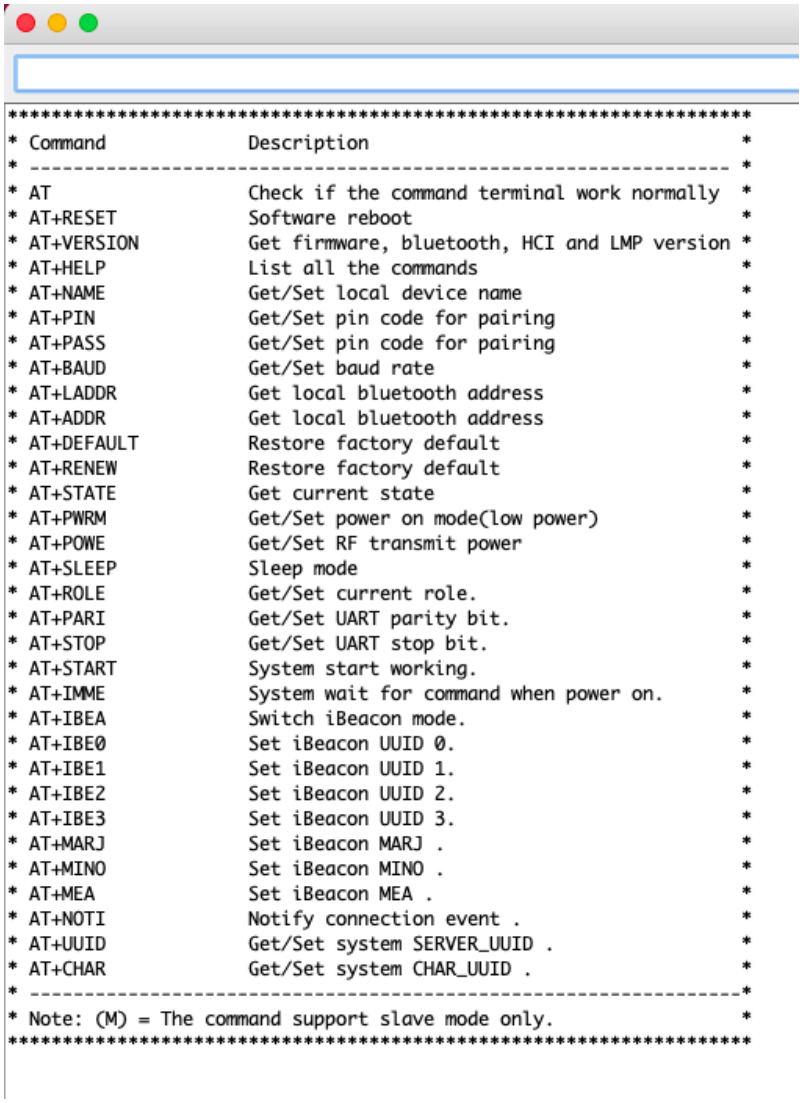

Figura 2.11: Lista de Comandos AT.

Como observamos obtenemos múltiples comandos que nos ayudarían a configurar nuestro módulo a nuestro antojo. Una de las cosas mas básicas que se ha realizado es cambiar el nombre del dispositivo, el cual se muestra a la hora de realizar cualquier búsqueda Bluetooth, su nuevo nombre visible para otros dispositivos será *BLE_SENSOR*. A continuación lo que vamos a hacer es a través de una aplicación

de escritorio del ordenador llamada *LightBlue* buscar nuestro dispositivo y ver cual es la configuración que usaremos del mismo. Buscando en dispositivos cercanos podemos encontrar nuestro módulo como vemos a continuación.

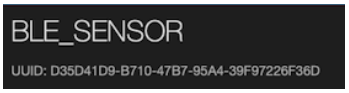

Figura 2.12: Dispositivo BLE Encontrado.

Podemos observar el contenido de los datos que publica el dispositivo para ser visible a los demás, lo que antes hemos llamado como *Advertisement Data*. Observamos también la información general que nos proporciona el módulo.

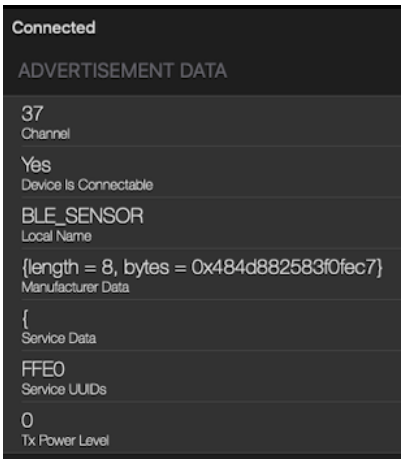

Figura 2.13: Contenido del *Advertisement Data*.

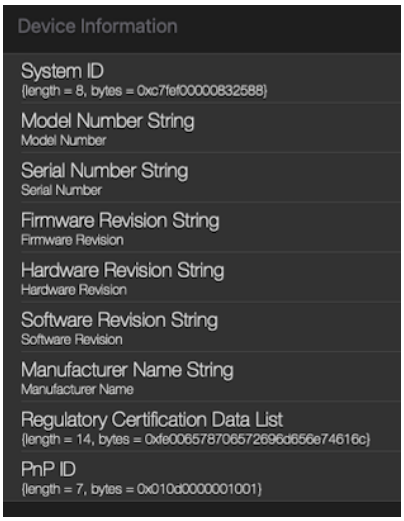

Figura 2.14: Información General del Módulo.

Finalmente podemos obtener la descripción del servicio 0xFFE0 que a su vez contiene la característica 0xFFE1, la cual, como podemos apreciar en la siguiente imagen, contiene como valor una cadena de caracteres vacía y diferentes propiedades con las que podemos manejar en ella, podemos escribir en ella, leer datos que tenga escrito y otras variantes como notificar un cambio o escribir sin espera de respuesta.

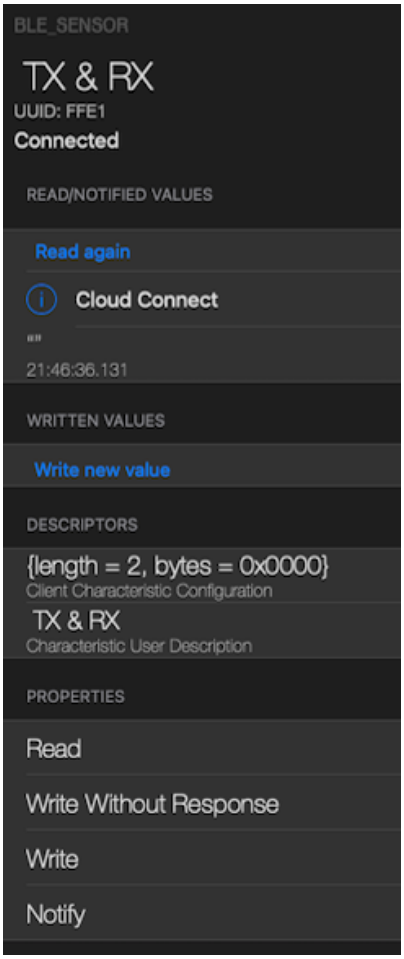

Figura 2.15: Estructura de Servicio y Característica del HM-10.

2.4. Descripción Comunicación MQTT.

En nuestro proyecto, tenemos la capacidad de administrar los datos que recibimos de todos y cada uno de los sensores en nuestro cliente BLE, el microcontrolador WiPy.Si este último tiene la opción de conectarse a una red WiFi, los datos recibidos serán reenviados a través de MQTT a una cola o *topic* previamente configurado.

MQTT son las siglas de *Message Queing Telemetry Transport*, siendo este un protocolo de comunación M2M, máquina a máquina, muy popular entre dispositivos IoT gracias a su simplicidad y poco consumo de recursos. Su funcionamiento se basa en lo que se conoce como pub-sub, donde unos dispositivos son publicadores

de mensajes y otros subscriptores. Estos dispositivos tienen que conectarse a un servidor que centraliza la comunicación conocido como *broker*. Los mensjes que un publicador envia al broker se dispone en lo que hemos llamado anteriormente como topic, al mismo tiempo otro dispositivo puede subscribirse a dicho topic y el broker se encarga de que le llegue dicho mensaje.

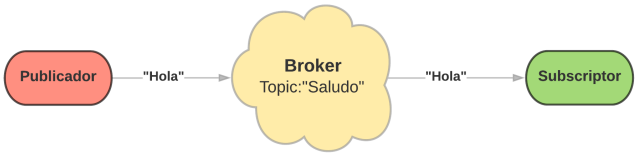

Figura 2.16: Ejemplo Comunicación MQTT.

Siguiendo este protocolo los clientes establecen comunicación con el broker mediante TCP/IP usando por defecto el puerto 1883. El cliente manda un mensaje de conexión con la identificación necesaria, como usuario y contraseña, y el broker contesta con un mensaje de recepción indicando si la conexión ha sido exitosa o no. Para subscribirse a un topic obtenemos la misma estructura pues el cliente tendrá que mandar un mensaje de suscripción o para darse de baja de dicho topic, siendo contestado por el broker con otro mensaje de vuelta. Para encolar un mensaje en un topic, el cliente tendrá que mandar una orden de publicar el mensaje al broker conteniendo este mensaje el topic y el contenido del mensaje a ser publicado. Como la conexion MQTT implica que la comunicación entre el cliente y el servidor deben estar activas, los clientes envían periodicamente mensajes para comprobar si esta sigue activa pues son contestado con el servidor, estos son conocidos como *PINGREQ* y *PINGRES*.

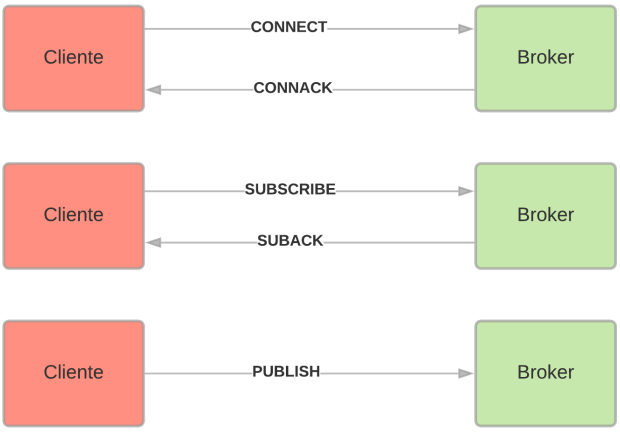

Figura 2.17: Ejemplo Mensajes MQTT.

Una de las características mas interesante es que dispone de distintos niveles de robustez del envío de mensajes frente a fallos, se conoce como QoS, *Quality of Service*, cada nivel superior de QoS implica un mayor intercambio de mensajes de

verificación y por lo tanto mas carga de procesamiento en el dispositivo, por lo que se deberá elegir en función de los requerimientos de robustez. Estos consisten en los tres siguientes tipos posibles.

- **QoS 0.** El mensaje es enviado como mucho una vez por lo que si falla puede no ser recibido.
- **QoS 1.** El mensaje se envía múltiples veces hasta que el cliente es notificado con la recepción del mensaje, esto implica que el suscriptor reciba multitud de mensajes.
- **QoS 2.** El nivel mas robusto garantiza la correcta recepción del mensaje y solamente una vez.

2.4.1. Broker de Adafruit.

Ya que conocemos lo básico de la comunicación por MQTT explicamos cual es el broker que hemos usado en el proyecto y explicaremos un poco acerca de su funcionamiento.

El broker usado es el mismo que aconseja la documentación de Pycom la empresa desarrolladora de WiPy, este se llama IO Adafruit, y como podemos conocer por su nombre es propiedad de la empresa Adafruit. Este broker lo podremos usar gratis con funciones limitadas pero que nos permite tener una primera toma de contacto con el protocolo MQTT.Para hacer uso del mismo debemos crearnos una cuenta en su sitio web https://io.adafruit.com/ una vez tenemos creada la cuenta podemos observar multiples opciones

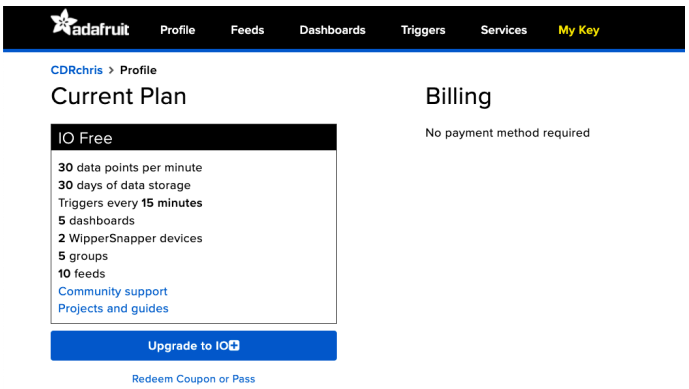

Figura 2.18: Perfil Personal Broker IO Adafruit.

Para poder tener topics a los que publicar mensajes en el broker, lo que tenemos que hacer es crear nuevos feeds, tal y como los dos que tengo creado como podemos ver en la siguiente imagen. Una vez tenemos los topics creados, el broker nos permite

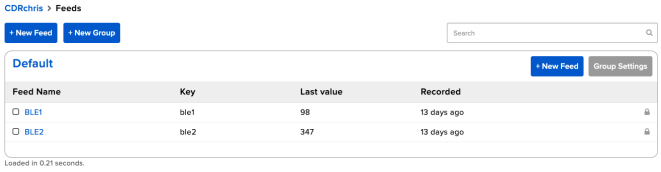

Figura 2.19: LIsta de Feeds IO Adafruit.

una configuración muy interesante y es que desde su misma web podemos crear gráficas personalizadas por cada uno de los feeds creados, es decir, podemos ir viendo en esta web directamente la evolución de las medidas que tomarian todos y cada uno de los sensores que tengamos en nuestra red, pues cada uno publicaría en un topic y podríamos tener su evolución. Finalmente para poder conectarnos desde un cliente

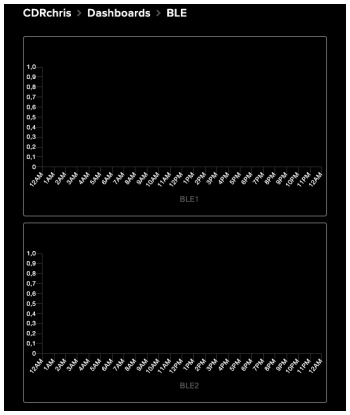

Figura 2.20: Dashboard Broker IO Adafruit.

MQTT externo como va a ser nuestra WiPy, lo que necesitamos son los siguientes datos de configuración.

- **Host.** Indica la dirección del broker a conectarnos en nuestro caso, io.adafruit.com.
- **Puesto** Como nuestra conexión no va usar ningún tipo de encriptación, usamos el puerto 1883 por defecto.
- **Usuario** En este caso necesitaremos indicarle nuestro usuario con el cual nos registramos en el broker.
- **Contraseña** Para que un cliente externo se conecte tenemos que generar una contraseña específica en el apartado API Key, no es la misma que la de nuestra cuenta
- **Topic.** Una vez tenemos creados los feeds, para acceder a publicar o leer de ellos su nombre serán del tipo siguiente: üsuario/feeds/"nombre del topic"

2.5. Implementación WiPy y MicroPython.

Una vez hemos hablado de las tecnologías que usamos para comunicarnos con los datos que nos proporcionan los sensores, así como los transmitimos al recibirlos, vamos a hablar con mas detalle en esta sección todo lo relacionado con el servidor BLE que centralice toda nuestra red de monitorización, en este caso la elegida es ela placa WiPy.

2.5.1. Microcontrolador WiPy.

El micrcontrolador que se eligió como cerebro principal en el proyecto no podía ser un microcontrolador de bajas prestaciones pues nos limitaría demasiado el alcance del mismo, así pues se optó por WiPy, el cual obtiene multiples ventajas en el apartado de las comunicaciones pues esta pensado para su integración con WiFi y Bluetooth, perfecto para lo que se le necesita en este proyecto.

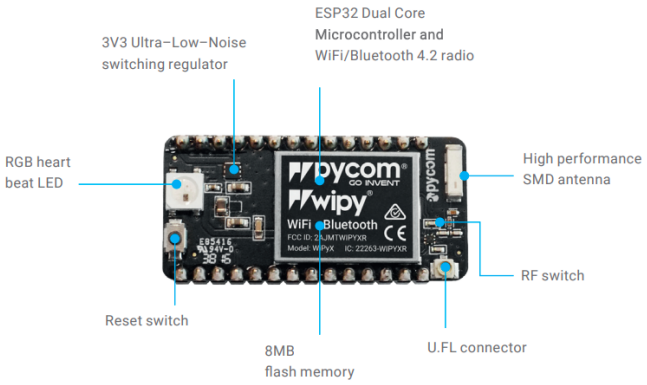

Figura 2.21: WiPy 3.0.

Esta placa esta basada en el microprocesador, ampliamente conocido en el mundo del IoT, por sus ventajas de conectividad IoT, ESP32 y esta fabricada en este caso por la compañía Pycom. Lo que nos encontramos en cuanto a especificaciones dentro de la misma son las siguientes.

- **CPU.** Incluye un microprocesador Xtensa de doble núcleo y 32 bits, dicho procesador contiene un acelerador de cálculo para punto flotante, así como permite la implementación multihilo de python. Además de la cpu podemos enconrar un co-procesador de bajo consumo utilizado para monitorizar los pines de entrada-salida, los convertidores analógico digital y la mayoría de periféricos internos cuando este se encuentra en modo de ultra bajo consumo.
- **Bluetooth.** Integra la capacidad de conectarse mediante Bluetooth clásico o los protocolos de comunicación BLE, los cuales son los que usaremos en el proyecto.
- **WiFi.** Capacidad para conectarse a redes WiFi 802.11b/g/n con un ancho de banda de 16mbps

- **RTC.** Su reloj de tiempo real interno es funciona a una frecuencia de 150 kHz.
- **Seguridad.** Soporta los protocolos de seguridad SSL/TLS, así como WPA.
- **Memoria.** Como memoria RAM tiene capacidad de 520 KB en conjunto con una memoria pSRAM de 4MB. Su almacenamiento interno llega hasta los 8MB.

A continuación mostramos la estructura interna que tiene nuestra placa, donde podemos observar sus periféricos internos y como se comunican con el núcleo central de la misma.

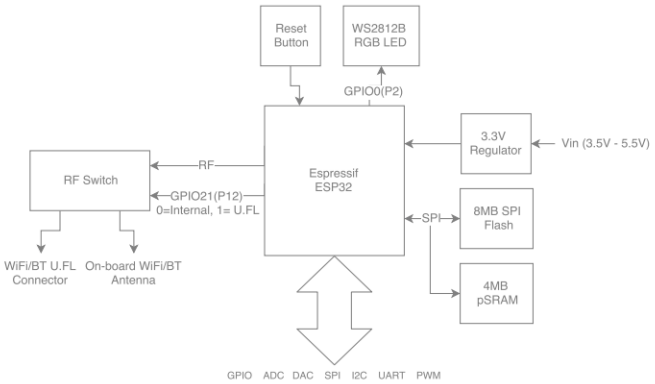

Figura 2.22: Estructura Interna de Periféricos.

Para usar dicha placa vamos a hacer uso de una placa de expansión muy útil para el desarrollo de la misma. Se trata de otro módulo donde integraremos la WiPy ofreciéndonos más posibilidades con ella. Nos proporciona un conector para instalar una batería externa, una entrada micro USB para conectarnos desde el PC, ya sea para programarla o para usar su interprete de Python REPL. También podemos observar que incluye una entrada de micro SD la cual nos sirve para escribir o leer ficheros de la misma, cumpliendo así unos de los requisitos iniciales del proyecto.

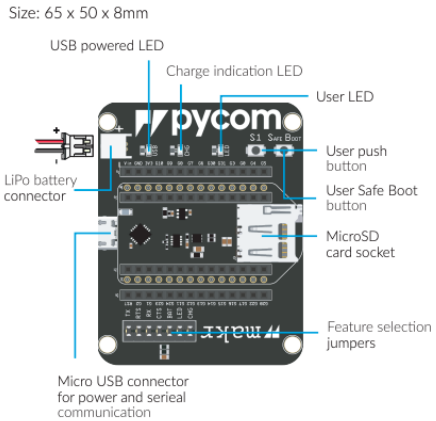

Figura 2.23: Placa de Expansión WiPy.

2.5.2. MicroPython en WiPy.

Conocemos a MicroPython como un lenguaje de programación optimizado para ciertos microcontroladores, dicho lenguaje es una implementación del famoso Python 3, y esta escrito en C de ahí su compatibilidad en sistemas embedidos sin la necesidad de un ordenador con el interprete original de Python. MicroPython es a su vez un compilador que traduce el código en Python que nosotros programamos a código de bajo nivel entendido por el micro, contiene un interprete de Pyhton a tiempo real conocido como (REPL) que se ejecuta usando el hardware del controlador, gracias a lo que hemos comentado anterioremente. Muchas librerías básicas de Python se encuentran disponibles para ser usadas integrando a la vez muchas librerías propias, las cuales nos dejan interactuar a bajo nivel con los periféricos que incluya el microcontrolador.

Figura 2.24: Logo de MicroPyhton.

Para el uso de MicroPyhton en nuestra placa WiPy para su programación se ha usado el IDE Visual Studio, en el que para que todo funcione perfectamente, debemos instalar un plugin llamado Pymakr. Este plugin tiene un funcionamiento bastante sencillo pues una vez instalado, obtenemos las mismas opciones que obtendríamos en cualquier IDE orientado a microcontroladores.También obtenemos una consola específica para nuestra placa donde al conectarla, esta se comunica con el puerto serie del USB del ordenador y de la misma WiPy mediante el micro USB disponible en la placa de expansión que hemos descrito anteriormente, obtendríamos directamente el interprete de MicroPython que se ejecuta en tiempo real en el micro pudiendo lanzarle comandos directamente sin compilar ni grabar dicho código en la memoria del mismo. A continuación podremos ver como se ejecutan ordenes sencillas en el REPL de WiPy

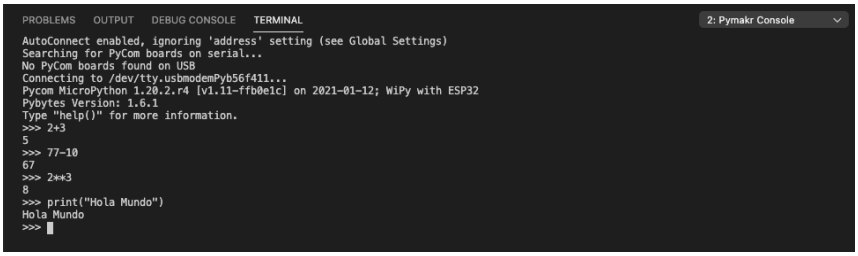

Figura 2.25: Ejemplo REPL MicroPython.

A continuación debemos de comentar como se estructura un proyecto en MicroPython, pues en su ejecución todos los programas en este lenguaje siguen unas normas.

De primeras tenemos que todos los archivos y carpetas que tengamos en nuestra ruta de proyecto en VS Code se copiaran en la flash de WiPy, siempre y cuando estos archivos se encuentre entre las extensiones permitidas en el archivo de configuración del proyecto *pymakr.conf*. La estructura básica de ejecución del proyecto se divide en dos archivos.

- **boot.py.** Se trata del primer conjunto de instrucciones que se lanzaran a la hora del encendido de WiPy, dicho archivo es lanzado en tan solo 150 microsegundos desde el encendido de la placa.
- **main.py.** Archivo donde encontramos la ejecución principal del proyecto, en caso de ser una ejecución cíclica como la mayoría de los microcontroladores aquí nos encontraremos con el bucle infinito.

Como podemos observar esta estructura de funcionamiento es la misma que en una gran de dispositivos comerciales que son usados para el desarrollo en microcontroladores, entre ellos Arduino, en el que obtenemos similitudes entre la función *setup* y el archivo *boot.py*, así como con la función *loop* y *main.py* en MicroPython.

2.5.3. WiPy como Servidor Web.

Una de las funciones principales que se necesita que la placa WiPy cumpla es la de poder funcionar como un servidor web para poder comunicarse con el a través de una página o que ella misma nos sirva la página donde podremos visualizar los datos de la red de sensores. A continuación nos vamos a centrar en como se ha conseguido esto.

Figura 2.26: Librería MicroWebSrv2.

Para conseguir dicho objetivo se ha usado la librería *MicroWebSrv2* creada por el usuario de la comunidad "jczic". Dicha libreria convierte nuestro microcontrolador en un potente servidor web con multiples características, expondremos algunas de ellas.

- Creada con una estructura puramente asíncrona lo que nos permite procesar varias peticiones a la vez con gran velocidad.
- Uso de multihilo para paralelizar procesos simultáneos.
- Implementa SSL/TLS permitiendo conexiones seguras en modo https.
- Procesa muchos tipos de peticiones HTTP como GET, HEAD, POST, PUT, DELETE, OPTIONS, PATCH.
- Envía y recibe objetos JSON, creando así una estructura del estilo API REST.

Se entrará mas en detalle cuando hablemos del flujo que sigue la ejecución del código pero nos vamos a centrar actualmente en que son los objectos JSON y que es eso de una estructura API REST.

JSON son las siglas de *JavaScript Object Notation* y son usados en el proyecto para mandar y recibir las respuestas al servidor. Podemos decir que se trata de una estructura de archivo en el que el documento se encuentra atrapado entre llaves ""donde cumple la estructura Clave/Valor, es decir observaremos unos campos con su nombre entre comillas, seguido de dos puntos y del valor de dicho campo. El valor puede ser varios tipos de datos como cadenas de caracteres, números enteros o incluso otro objeto JSON dentro.

Describiremos a continuación la estructura que seguirá el servidor implementado con esta librería. Como hemos comentado implementa una estructura del estilo API REST. Esto quiere decir que en nuestro servidor lo que definiremos son rutas, o lo que se conoce como *endpoint* con un nombre. Este nombre aparecerá al final de la url de acceso al servidor precedido del símbolo '/', quedándonos lo que sería *url/endpoint* y a todo ese conjunto es al que le mandamos la petición. Cuando el servidor detecta una petición a dicho endpoint, procederá a ejecutar el código que hemos programado dentro de esa ruta. Como entrada a dicho código desde el cliente le llega un objeto JSON, cosa que en python lo podemos extrapolar a los tipos de datos llamados "diccionarios", finalmente una vez después de la ejecución de la lógica que queramos podemos contestar al cliente con otro objeto JSON. La respuesta puede estar formada por un código HTML que tengamos guardado en la memoria Flash de WiPy, lo que implicaría que podemos mandarle una petición a *url/endpoint1* y recibir de respuesta en el navegador la página web estática codificada en HMTL. Esta estructura puede escalar formando múltiples endpoints que cada uno haga lo que necesitemos.

Una vez hemos comentado básicamente los conceptos necesarios para entender el funcionamiento de la solución propuesto hemos de completar dicha explicación exponiendo los modos de conexión WiFi que nos permite WiPy.Nos encontramos con que a la hora de utilizar dicha tecnología en la placa, tenemos dos modos de conexión.

- **AP.** Así es como se describe el uso de la red WiFi de la placa como *Access Point*. Configurando WiPy en este modo esta se convierte en una especie de

router donde podemos conectarnos, es decir, sirve una red WiFi a la que podemos conectarnos con usuario y contraseña, tal y como haríamos en nuestra red WiFi doméstica.

- **STA.** En este caso de conexión nos encontramos con el funcionamiento normal de un dispositivo, *Station* queriendo decir que la placa se conectará a una red preexistente teniendo acceso a internet, cosa que no ocurre en el caso anterior.

En el proyecto ambos tipos de conexión serán usados, explicaremos mas en detalle cuando y para qué cada uno de ellos, sin embargo hemos de comentar que en el modo de conexión AP, podremos configurar el nombre de la red que aparece para los demás dispositivos pudiendo conectarnos a ella y así lanzar peticiones a WiPy, con la direccion http://192.168.4.1/ y las distintas rutas que hemos definido con la librería *MicroWebSrv2*.

Capítulo 3

Descripción Servidores BLE.

En el siguiente capítulo entraremos en detalle en todo lo relativo al conjunto del servidor BLE que hemos organizado, como descripción de componentes, conexiones y funcionamiento.

3.1. Sensores Usados.

Como ya se ha comentado el objetivo de la red de sensores será distribuir módulos en los que podamos leer la cantidad de CO_2 que existe en el ambiente, para eso se han creado módulos que llamaremos servidores a partir de ahora, pues nos sirven los datos, compuestos por un microcontrolador, un módulo BLE HM-10 y un sensor de CO_2.

3.1.1. MQ-135.

El primero de los sensores que descubrimos para leer gases en el ambiente fue el conocido MQ-135. Este sensor es muy conocido en cantidad de proyectos IoT pues es usado para equipamientos de control de la calidad de aire en diferentes estancias, ya que es capaz de detectar gases como NH_3, NO_x, Alcohol Etílico, Benceno,CO_2. Entre sus principales características que se consideraron atractivas para usarlo fueron.

- Amplío rango de detección de gases, pues se podría usar no solo para un gas si no para la concentración de muchos tipos así como la calidad del aire.
- Rápida respuesta y alta sensibilidad, pudiendo obtener medidas de forma ágil así como detectar cambios de concentración repentinos.
- Diagrama de conexión muy sencillo pudiendo implementarse rápidamente en cualquier microcontrolador.
- Precio muy reducido, lo que apoyaría a distribuir muchos de ellos en diferentes lugares de una misma habitación incluso.

Como podemos ver en la siguiente imagen el sensor se encuentra instalado en un módulo listo para ser integrado. En el se puede apreciar una especie de tornillo de

plástico el cual nos sirve para modificar la sensibilidad del mismo en el uso de su salida digital.

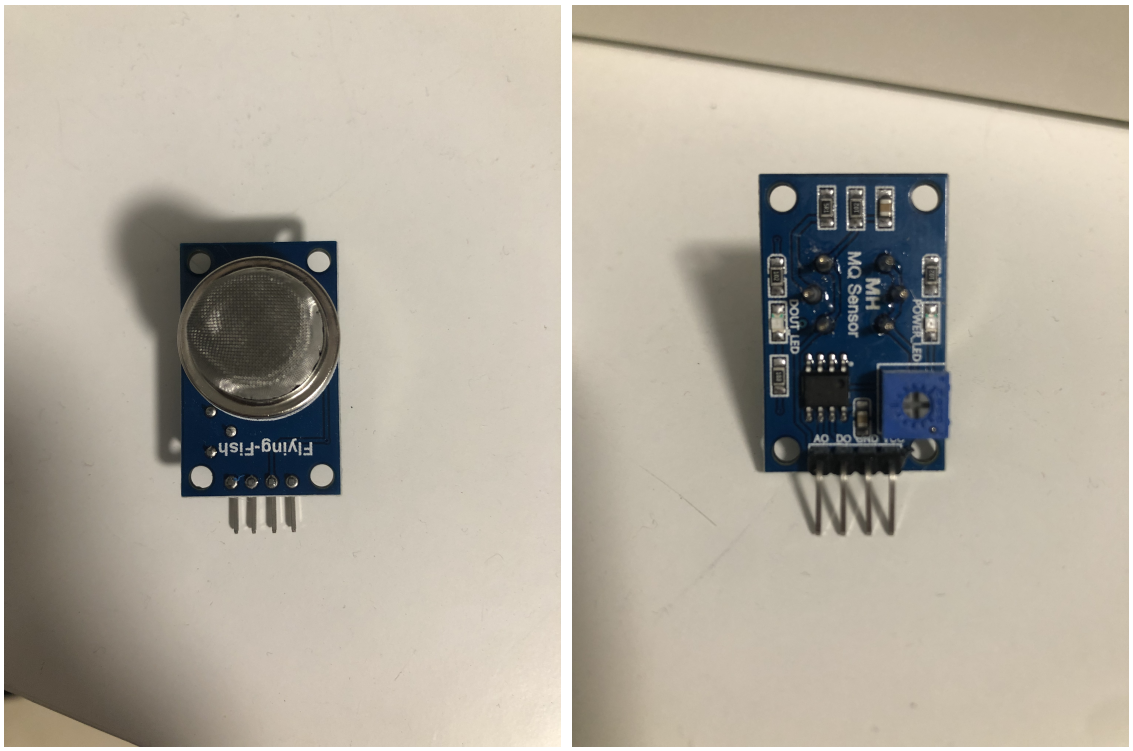

Figura 3.1: Módulo Sensor MQ-135.

Pin	Uso
Vcc	Alimentación a 5V.
GND	Conexión a tierra.
A0	Salida analógica para lectura de datos.
D0	Salida digital indicando 1 o 0 según sensibilidad ajustada

Cuadro 3.1: Salidas Sensor MQ-135

Como apreciamos en sus pines de salida podemos usar el sensor de dos formas. Si usamos su pin digital este funciona como una alarma que nos indica que el sensor esta midiendo la cantidad de gases en el ambiente que nosotros hemos ajustado con el potenciómetro. Bien podemos usarlo directamente midiendo su salida analógica lo que nos dará un valor cada vez que lancemos la orden de lectura desde el micro-controlador.

Sin entrar mucho mas en detalle este sensor ha resultado ser un fiasco a la hora de conseguir una lectura del gas $C0_2$ en el ambiente ya que no sirve para el fin que lo queríamos como veremos en las pruebas realizadas, la concentración de múltiples gases hace sus medidas inservibles para nuestro fin.

También hay que destacar de este sensor y es su cantidad de réplicas que se venden en el mercado lo que hace que no sea fácil encontrar una que sirva para medir nada. Las características del producto indican que el transductor tiene una resistencia de carga de 20 $k\Omega$ cosa que al final no ocurre ya que los fabricantes de dicho modulo integran resistencias de carga de 1 $k\Omega$ teniendo medidas que a la práctica no nos sirven para nada. Es por eso que buscamos una alternativa a dicho sensor.

3.1.2. CCS811.

La solución al imprecisión del MQ-135 encontramos otro tipo de sensor de CO_2, el llamado CCS811. Se trata de un módulo fabricado por la empresa Keyestudio, usado para medir la calidad del aire así como la concentración de dióxido de carbono en el ambiente, usando principalmente el chip CCS811. Dicho chip es un sensor de gases de ultra bajo consumo que es capaz de detectar una amplia variedad de gases de compuestos orgánicos, los conocidos como TVOCs, incluyendo el equivalente del dióxido de carbono,eCO_2. La concentración de dicho gas puede ser medido entre una concentración de 400 hasta 29206 ppm.

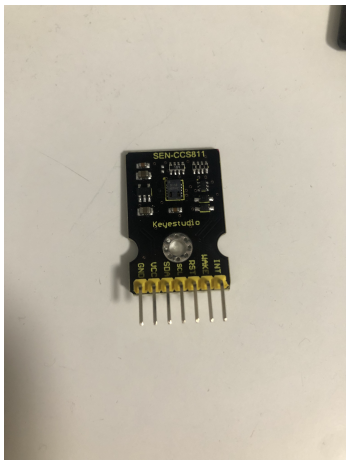

Figura 3.2: Sensor CCS811.

- Es alimentado a 5V como la mayoría de módulos usados en este tipo de proyectos.
- Cuando nos encontramos midiendo la corriente de trabajo es alrededor de 30mA haciendo que su máximo consumo de potencia se sitúe hasta los 60mW.
- La capacidad de medida de CO_2 como hemos comentado se sitúa entre 400 y 29206 ppm, así como el rango de medida de TVOCs está entre 0 y 32768ppm

Como podemos apreciar en sus pines de conexión este módulo integra un tipo de conexión diferente, pues en el anterior simplemente leíamos el valor en su pin de salida analógica, pero en este caso, tenemos un protocolo de comunicación I2C. En este tipo de comunicación contamos con dos elementos conocidos como maestro

Pin	Uso
Vcc	Alimentación a 5V.
GND	Conexión a tierra.
SDA	Pin de datos I2C.
SCL	Pin del reloj I2C.
RST	Si conectamos a tierra se reinicia el módulo.
WAKE	Debe estar conectado a tierra para comunicarnos.
INT	Pin interrupción que detecta una medición disponible.

Cuadro 3.2: Salidas Módulo CCS811.

y esclavo, donde estos dos se comunicarán a través de dos buses. Estos buses son conocidos como SDA *Serial Data* y SCL *Serial Clock*, es decir tenemos un bus de datos y otro bus para el reloj serial.

- **Maestro.** Es el encargado de iniciar y parar la comunicación así como de controlar el reloj. La información viaja a traves de un solo cable de datos por lo que dos maestros no pueden ser conectados a un mismo bus SDA. Pueden funcionar como transmisor o receptor de datos.
  - Inicia la comunicación.
  - Envía 7 bits de dirección.
  - Genera 1 bit de escritura o lectura.
  - Envía 8 bits de dirección de memoria o transmite 8 bits de datos.
  - Confirma la recepción o no de los datos.
  - Finaliza la comunicación
- **Esclavo.** Suele ser el sensor el cual suministra la información al maestro. Puede actuar también como transmisor o receptor.Al contrario que el maestro, no puede generar la señal SCL.
  - Envía 8 bits de datos
  - Envía confirmación de recepción.

3.2. Diagramas de Conexión.

Habiendo explicado que sensores han sido probados en el proyecto dejamos constancia de los diagramas de conexión que se han usado conjunto con el microcontrolador Arduino. Como veremos a continuación se incluye la conexión del módulo HM-10, conectando TX al pin 2 y RX al pin 3, siendo el puerto serie de Arduino a la inversa para una conexión satisfactoria. Los conjuntos que aquí formamos serian los servidores BLE que alimentados en cualquier lugar mandan datos al Cliente BLE, WiPy.

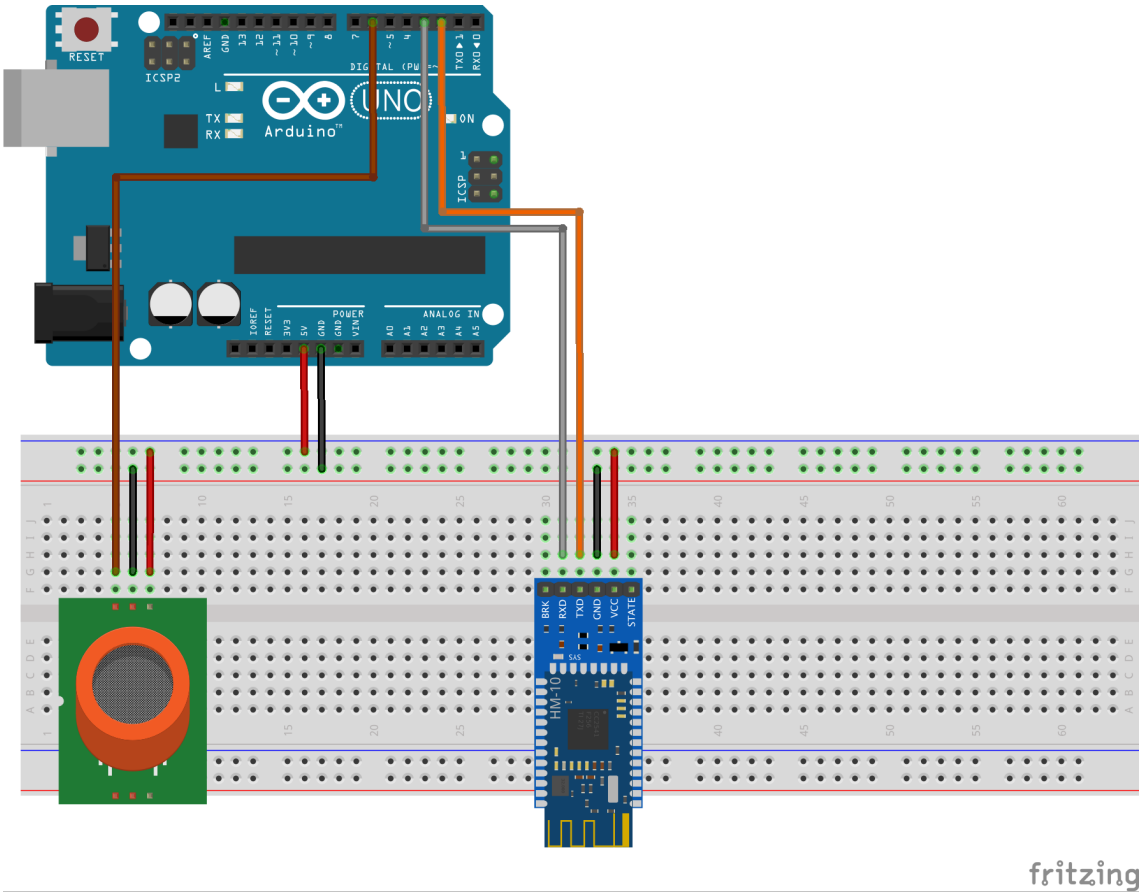

Figura 3.3: Diagrama de conexión MQ-135.

Pin	Arduino
Vcc	Alimentación a 5V.
GND	Conexión a tierra.
A0	Pin A6 .

Cuadro 3.3: Conexiones Arduino MQ-135.

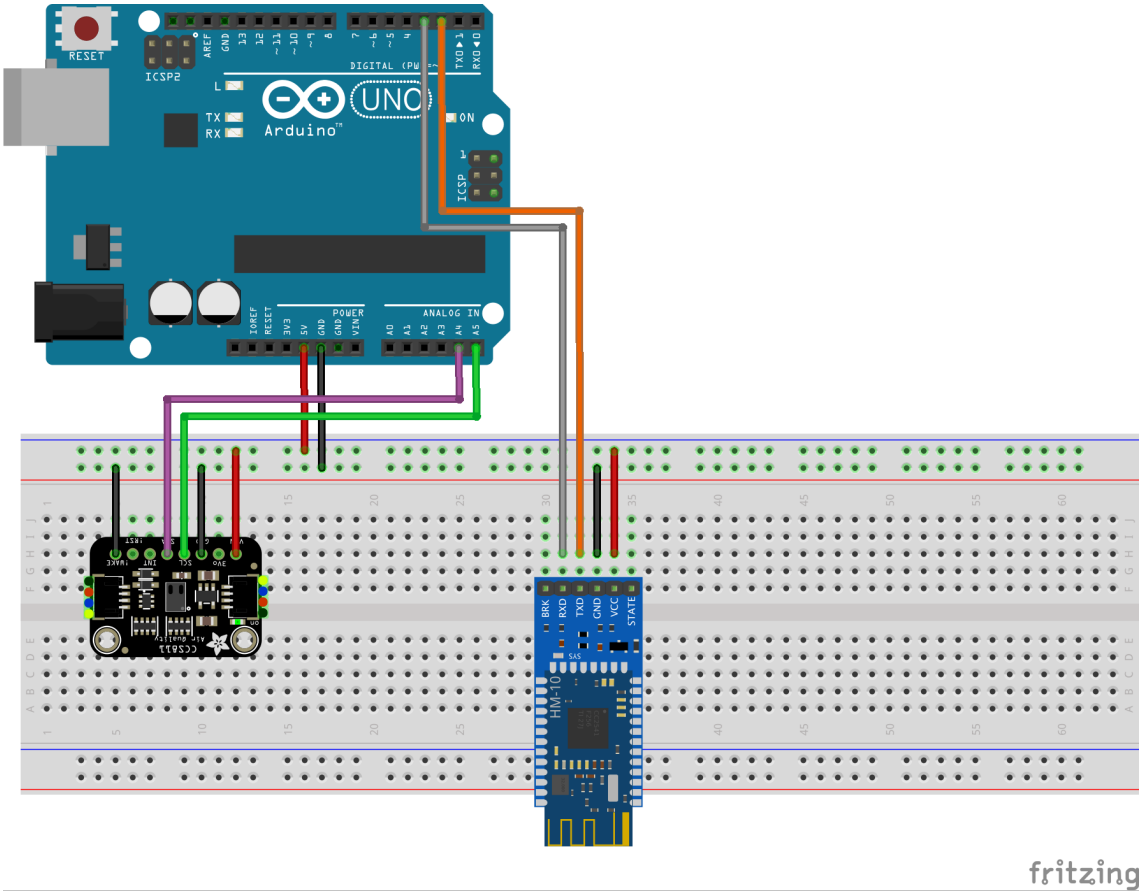

Figura 3.4: Diagrama de conexión CCS811.

Pin	Arduino
Vcc	Alimentación a 5V.
GND	Conexión a tierra.
SDA	Pin A6.
SCL	Pin A5.
WAKE	Conexión a tierra.

Cuadro 3.4: Conexiones Arduino CCS811.

3.3. Programación Servidor BLE.

Teniendo los conjuntos Sensor-Arduino-BLE conectados procedemos a explicar como se ha programado el microcontrolador para enviar los datos de los sensores. Como tenemos dos sensores diferentes implementados tendremos dos tipos de programas diferentes.

3.3.1. Lectura y Envío con MQ-135.

Como ya sabemos de su descripción dicho sensor es muy fácil de implementar pues solo estaremos usando su salida analógica. Como ya comentamos, dicho sensor no resulta ser válido en el desarrollo que buscábamos por ello no incidiremos demasiado en el mismo.

Para comenzar a utilizar dicho sensor el fabricante recomienda realizar un proceso de prequemado. Este proceso consiste en dejar conectado a 5V el sensor durante al menos 24 horas, al basar su medida en el valor de una resistencia que varía según su valor, este proceso acaba con las impurezas que esta pueda tener de su proceso de fabricación gracias al calor que genera. Tenemos entonces que este sensor lo vamos a dejar definido en el diagrama básico global de lo que queremos conseguir en dicho servidor BLE, donde la lectura de los datos del mismo se realizan simplemente leyendo la salida analógica del módulo conectado y enviando los datos mediante el puerto serie configurado al HM-10.

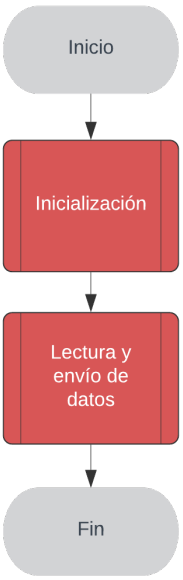

Figura 3.5: Flujo Global de Lectura en Arduino.

3.3.2. Lectura y Envío con CCS811.

Como en este caso tenemos el producto implementable completo, pasaremos a una descripción más amplía del proceso que seguímos en dicho programa.

En principio la idea es la misma, ya que el proceso será exportable a cualquier sensor.Leemos datos y lo escribimos en la característica del módulo BLE conectado que ya será leída por el cliente. En este caso tenemos que dividir el flujo del microcontrolador en dos. Primeramente contamos con la inicialización del mismo así como del sensor, donde en este caso, al comunicarse por I2C con el Arduino hemos de configurarlo previamente. Los pasos que se siguen están detallados en el siguiente diagrama.

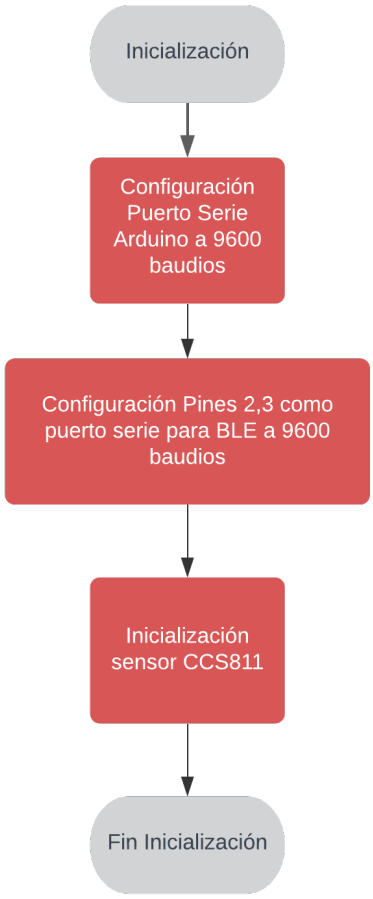

Figura 3.6: Inicialización Arduino.

Una vez configurado todo el microcontrolador entra en un bucle infinito de lectura y envío de datos, este solo hace de *bypass* entre el sensor y el HM-10. Hay que aclarar que el programa esta preparado para que podamos introducir manualmente datos en la característica BLE, quiere decir que si mandamos un dato por puerto serie o

desde otro dispositivo BLE escribimos un nuevo dato en la característica esto será impreso en la salida del serial de Arduino.

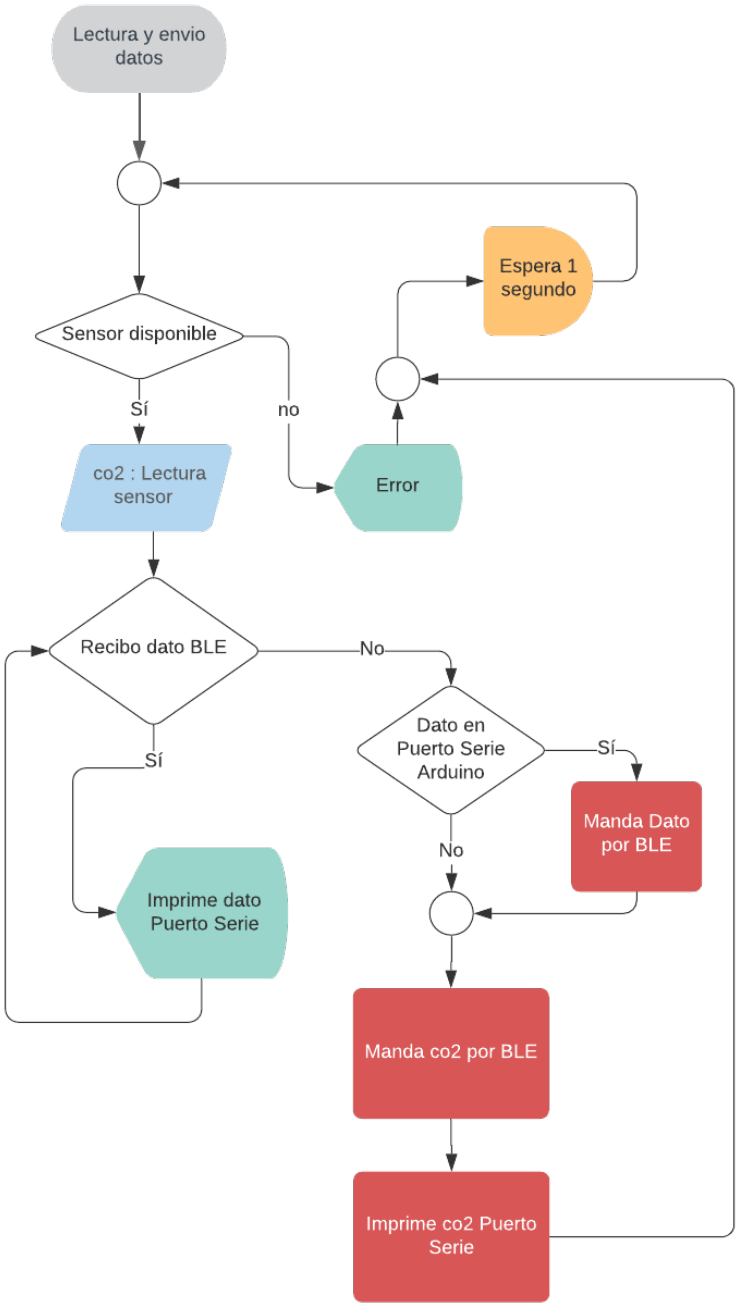

Figura 3.7: Bucle Infinito Arduino.

Capítulo 4

Desarrollo WiPy como Cliente BLE.

En el siguiente capítulo nos centraremos en el software necesario que hemos desarrollado para hacer funcionar la placa WiPy según los requerimientos del proyecto.

4.1. Librerías Desarrolladas.

Para completar las funcionalidades que nos ofrece el microcontrolador hemos tenido que desarrollar un par de librería para controlar los procesos que esta realiza tal y como se quiere. Dichas librerías serán codificadas como clases estándares de Python en la que haremos uso de las funciones integradas, para finalmente alojar dicho archivo en la carpeta *lib* en la ruta del proyecto.

4.1.1. Librería BLE.

En primer lugar contamos con una librería que ya nos integra Pycom en la placa para controlar las distintas conexiones inalámbricas, llamada *network*, en ella podemos encontrar los métodos necesarios para controlar las conexiones Bluetooth o incluso la conexión a WiFi. Esta librería nos permite la realización de los pasos básicos necesarios para implementar las comunicaciones, pero no el control del proceso como nosotros queremos, es por eso que en este caso hemos creado una librería llamada *BLE_process.*

Se trata entonces de la librería que nos aporta el funcionamiento principal del dispositivo que es la de funcionar como un cliente BLE, es por eso que entraremos más en detalle en su composición. Dicha clase necesita la inicialización de los siguientes atributos para su correcto funcionamiento.

- **mac** Guardará el valor de la dirección MAC del servidor BLE al que nos vamos a conectar.
- **sensor_config** Obtendremos un diccionario con toda la configuración necesaria para la conexión del servidor BLE correspondiente.
- **client_mqtt** Objeto que usaremos para enviar los mensajes recibidos mediante el protocolo de comunicación descrito anteriormente MQTT.

- **topic** Guarda el nombre de la cola a la que mandar el mensaje MQTT.
- **max_conn** Número entero que guarda el máximo de dispositivos al que nos vamos poder conectar.
- **ble_search_timeout** Tiempo máximo en segundos que nos pasaremos buscando el dispositivo objetivo.
- **reconnection_retries** Indica la cantidad de veces que vamos a intentar conectarnos a dicho dispositivo.
- **last_value** Último dato recibido por ese dispositivo.
- **connected_devices** Variable booleana que indica si tenemos conexión establecida con el dispositivo.
- **bt** Guardamos en otra variables la instancia de la clase Bluetooth que incorpora WiPy.
- **conn** Objecto de la conexión establecida con el servidor BLE.

Una vez conocemos los datos que necesitamos para ejecutar el proceso BLE que hemos desarrollado vamos a proceder a describir los flujos que siguen las funciones Integradas. Comenzaremos describiendo brevemente cual es su objetivo individual.

- **notification_callback** Este método estático es el que será ejecutado siempre que WiPy detecte un nuevo dato escrito en la característica del sensor BLE al que este conectado. Su única función será guardar el dato en nuestro fichero de datos, y si nos encontramos conectados a un broker MQTT enviar el dato al topic correspondiente.
- **connect_to_ble** Se trata de un método que simplemente envuelve a la función primitiva de conexión a dispositivo integrada. Tenemos que simplemente usaremos este método para conectarnos a un dispositivos BLE de dirección MAC conocida.
- **init_ble_search** Ahora nos encontramos con una lógica más complicada pero lo suficientemente sencilla para poder describirla con palabras. Observamos una función diseñada para buscar entre todos los dispositivos Bluetooth a nuestro alrededor, aquel que tengamos configurado.Esto lo hace gracias a lo que comentamos anteriormente sobre el .ªdvertising data"de los dispositivos BLE. La búsqueda tiene un limite de los segundos configurados y al encontrar nuestro dispositivo se devolverá lso datos de advertising, la dirección MAC encontrada y el nombre del dispositivo BLE.
- **main_ble_process** Como su nombre indica estamos ante la función principal que manejará el proceso de conexión a un dispositivo. Su estructura es más compleja así que la veremos mejor con un diagrama.

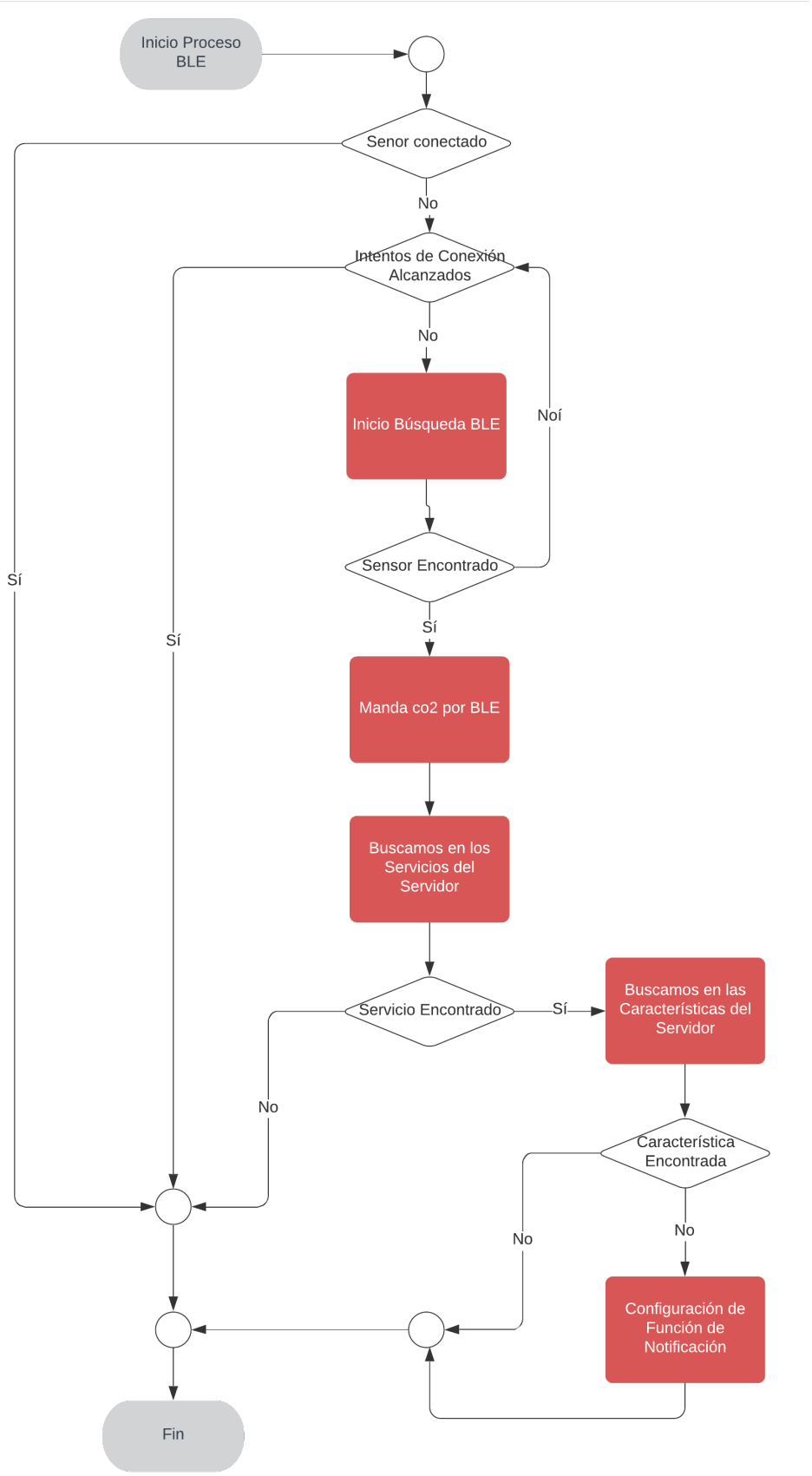

Figura 4.1: Diagrama Funcionamiento Proceso BLE.

4.1.2. Librería MQTT.

Para poder hacer uso de la comunicación MQTT a parte de estar conectados a una red WiFi necesitamos una librería que nos aporte las funciones a utilizar al igual que con el Bluetooth. El firmware de Pycom que lleva la placa WiPy integra una librería para dicha funcionalidad, aunque para ello hemos decidido usar la genérica de MicroPython *umqtt*. Esta no la podemos importar como de costumbre pues no viene integrada como hemos dicho, así que debemos descargarnos desde el repositorio oficial de MicroPython el archivo ümqtt.pyƶ ponerlo en el directorio de librerías del proyecto

Aún así se ha desarrollado una librería en forma de envoltorio de esta misma para hacer un uso mas claro del proceso. Esta librería la hemos llamado "mqtt_wipy". Como hemos hecho en la librería BLE, vamos a ver que variables necesitamos para el funcionamiento de la misma.

- **server** Host del broker MQTT.
- **port** Puerto del broker MQTT.
- **user** Nuestro usuario de la cuenta que tenemos en el broker.
- **password** Contraseña asociada a nuestro usuario.
- **client_id** Identificador único del cliente MQTT.
- **topic_list_sub** Lista de los topics a los cuales vamos a subscribirnos.
- **client** Instancia del objeto de cliente MQTT.

Como vemos la mayoría de variables que necesitamos crear son todas aquellas relacionadas con la conexión de nuestro cliente MQTT, que va a ser la placa WiPy, con el broker MQTT, que como vimos anteriormente se trata del broker disponible en io.adafruit.com. A continuación hemos incluido diferentes funciones que la totalidad de ellas carecen de un proceso a seguir, si no que son funciones que ejecutan una finalidad asecas.

- **suscribe_callback** Función estática que será lanzada cada vez que se lanzará cuando se detecte un nuevo dato en alguno de los topics que estamos subscritos que no vengan de la propia WiPy.
- **subcription_callback** En este método indicaremos la lógica de la función anterior como l que se va a lanzar en ese caso.
- **connect_client** Mediante esta función nos conectaremos al broker definido.
- **disconnection** Podremos desocnectarnos del cliente si queremos.
- **alive** Indica si la conexión MQTT se encuentra activa.

- **sub** Mediante ella nos subscribimos a la lista de topics configurados.
- **pub** Podemos publicar el mensaje que queramos en el topic que le mandemos.

Con todos estos métodos ya estamos en condiciones de usar MQTT en nuestro proceso.

4.1.3. Utilidades.

Para solventar algunas funciones hemos tenido que crear unos métodos nuevos, de los que iremos haciendo uso en el proceso.

Primeramente encontramos con unas funciones transversales básicas.

- **connect_wifi** Haciendo uso de esta función, indicándole el nombre y contraseña de la red a la que nos queremos conectar, conseguimos establecer conexión. que se va a lanzar en ese caso.
- **parse_config** Recogemos el JSON de configuración del proceso transformándolo a un diccionario de Python
- **wirte_config** Reescribimos el JSON de configuración que se encuentre guardado en la memoria flash de la placa.

Por último tenemos que solventar el requerimiento que pedía registrar los datos en una memoria externa microSD. Para ello nos hemos creado una clase a parte de Log, en la que contamos con tres casos.

- **info** La usaremos para llevar un registro de los pasos que va llevando el proceso. que se va a lanzar en ese caso.
- **data** Con ella guardaremos los datos que recibamos de los servidores BLE para poder llevar un registro sin necesidad de conexión.
- **clear** Eliminamos los datos que tengamos en los ficheros, tanto de log como de datos.

Como hemos podido ver el registro en de datos en la microSD se ha separado en dos tipos. Tendremos un fichero llamado "log.txt"para las indicaciones que queramos, y otro llamado "data.txt"para el registro de datos. Contemos o no con una conexión a internet los datos podran ser extraidos de la microSD y leido en los ficheros que hemos creado en la misma.

4.2. Inicialización del Proceso.

Como ya sabemos cual es la como procede la ejecución de un proyecto en MicroPython, observamos que el primer archivo en ejecutarse será el llamado "boot.py". Procedemos en este apartado a explicar que contiene dicho programa que hemos diseñado.Iremos viendo los pasos esenciales que sigue dicha lógica.

- Al comenzar dicho programa el primer paso que realizaremos será configurar el uso de la tarjeta microSD. Para ello montaremos la memoria externa para que pueda ser reconocida por el microcontrolador y borraremos los archivos de logs que se encuentren en ella, dejando los ficheros de extensión ".txt"listos para ser usados en esta sesión.

- Seguidamente declaramos un par de variables globales que nos servirán de ayuda en el proceso. Una de ellas será un flag de tipo booleano "start_main"la cual nos indicará cuando se ha acabado el proceso inicial de WiPy para dar paso al proceso principal. La otra se conocerá como çonfigz en ella guardaremos el diccionario de configuración que guardamos como tipo JSON en la flash de la placa.

- Configuramos una ruta, o *endpoint*, en la dirección /config.ªccesible mediante el verbo HTTP, POST. En esta ruta llamará el servidor entregándole la configuración que le proporcionemos a través de la web. Una vez teniendo estos datos actualizamos el fichero de configuración que ya hemos leído antes y lo escribimos en la memoria, poniendo el flag de inicio a verdadero.

- Configuraremos la red WiFi como modo de conexión AP, por lo que la placa creará su propia red pudiéndonos conectar a ella y siendo accesible para servirnos la web de configuración.

- Configuramos el micro como un servidor mediante la ya conocida librería MicroWebSrv2, haciendo que en su ruta por defecto, "host/"nos sirva la web preparada codificada en el archivo ïndex.html".

- Mientras el servidor este corriendo y el flag inicial este con un valor falso, dormimos durante 1 segundo el proceso.Cuando esto no se cumpla se acaba el proceso y arranca el archivo "main.py"

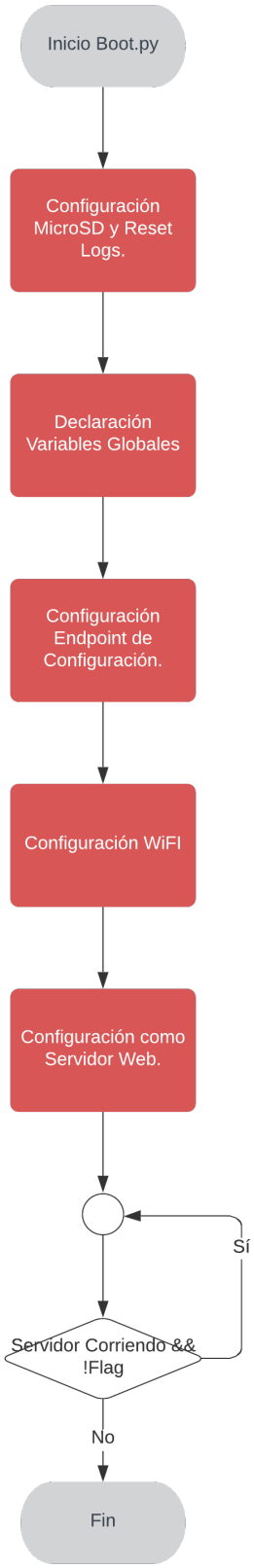

Figura 4.2: Diagrama del Proceso de Configuración.

4.3. Desarrollo del Proceso.

Habiendo acabado el proceso seguido en la configuración inicial, inmediatamente después, arrancará el fichero "main.py". En este fichero encontramos el desarrollo que se mantiene ejecutando el cliente BLE, en todo momento. Ya que tenemos dos tipos de modo de funcionamiento, se han separado en distintos archivos para tener bien estructurado el desarrollo, es por eso que el mismo fichero main solo implica una bifurcación para un tipo de proceso u otro según la configuración que le hemos dicho.

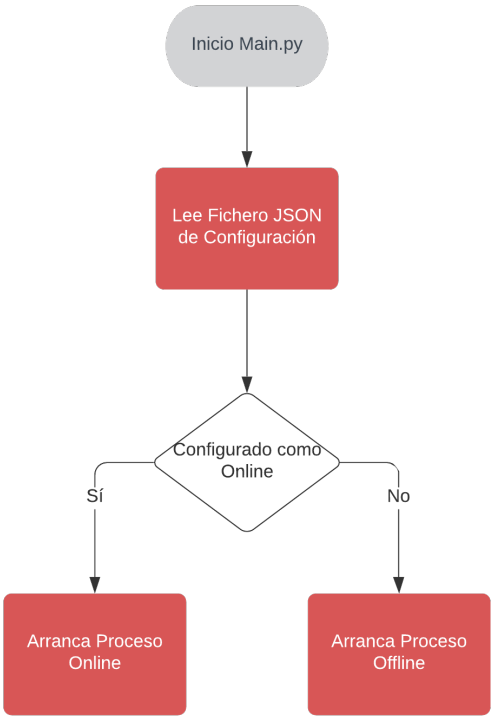

Figura 4.3: Diagrama del Fichero Main.

4.3.1. Funcionamiento con Internet.

En este apartado entraremos en profundidad las funcionalidades posibles y el proceso que sigue el cliente BLE si decidimos que este arranque su proceso Online. Este proceso implica tener una red WiFi externa que nos permita conectarnos a internet, por lo que sería lo ideal en lugares en los que esto ocurra como podría ser un edificio de oficinas o el interior de una casa.

Cuando elegimos el proceso Online, mediante la Web de configuración inicial, implica que no necesitamos conectarnos directamente a la placa mediante su dirección, si no que al tener acceso a internet no tiene sentido usar esta como servidor Web. Podemos concluir que en este proceso WiPy no nos servirá ninguna web directamente a nosotros, si no que los datos que recibamos del proceso BLE lo enviará a los topics correspondientes de cada sensor mediante MQTT. Por ello cuando elegimos Online tendremos que incluir también la configuración de nuestro broker MQTT. En este caso nos encontramos con los siguientes métodos.

- **wifi_connection** Este método será llamado para conectarnos a la red definida en la configuración, al cual como cabe de esperar le pasaremos el nombre y contraseña de la red.
- **mqtt_process** Función que encapsula el proceso a seguir para conectarnos satisfactoriamente por MQTT. En él simplemente crearemos el cliente y lo configuraremos con las funciones de la librería creada.
- **ble_service** Función a usar para arrancar el proceso principal BLE, en cada uno de las instancias de la clase BLE creadas en una variable global.
- **online_main** Función principal del proceso que se ejecutará indefinidamente. Cabe destacar que en ella configuramos una interrupción que cada 15 segundos vuelve a lanzar el servicio BLE, volviéndose a conectar con algún sensor con el que hubiera perdido la conexión.

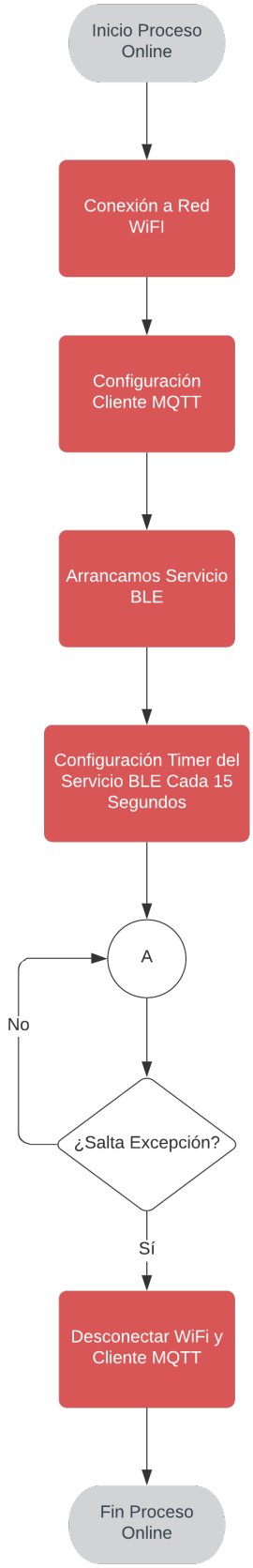

Figura 4.4: Diagrama del Proceso Online.

4.3.2. Funcionamiento sin Internet.

Cuando no disponemos de una conexión a una red Wifi externa que nos permita acceder a internet, siendo esto posible si queremos desplegar el proyecto en una zona rural como cultivos con otro tipo de sensores, o si simplemente no queremos que tenga acceso nuestros datos a internet. En este caso WiPy ejercerá de servidor Web pudiéndonos conectar a ella a través de la red que desplegará como punto de acceso y sabiendo su dirección. En este caso a través de una ruta determinada podremos observar unas gráficas de la evolución temporal que toman todos los sensores de nuestra red, gracias a las siguientes funciones.

- **Endpoint /fetchData"** A esta ruta será la que la misma web mandará una petición para recoger los últimos valores que ha recibido WiPy de todos y cada uno de los sensores configurados.
- **Endpoint /graphs"** Endpoint al que nos conectaremos para poder visualizar las gráficas de los sensores BLE.
- **ble_service** Mismo método que en el proceso Online.
- **init_web_server** Hace lo necesario para configurar WiPy como punto de acceso. También configura.
- **local_main** Sigue el mismo proceso que en el modo Online pero ahora no configuramos ningun client MQTT pues no usaremos dicha comunicación. La diferencia es que arranca el servidor al igual que haciamos en el "boot.py".

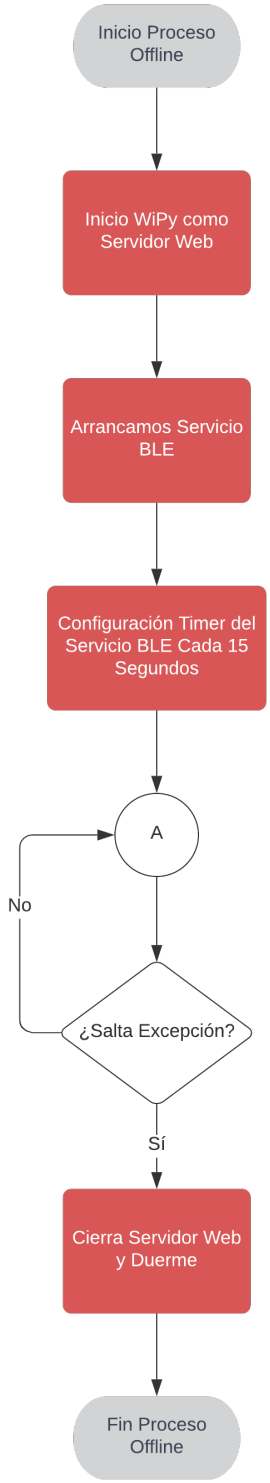

Figura 4.5: Diagrama del Proceso Offline.

4.4. Desarrollo Web

Una vez ya tenemos el conjunto de nuestra red de sensores enviando datos y a nuestro cliente principal recibiéndolos decidimos que la idea era plasmar dichos datos para ver su evolución en el tiempo, y para ello teníamos que diseñar distintas páginas web básicas.

Para ello por el diseño inicial del proyecto nos topamos con dos situaciones muy diferenciadas entre ellas, siendo el modo de conexión de WiPy el principal escollo. Si el funcionamiento permitía conectarse a una red WiFi, quiere decir que tenemos acceso a internet y los datos los tenemos que leer desde los mensajes MQTT que mandamos. Existen múltiples empresas en el mercado que permiten el desarrollo de dashboards para dispositivos IoT, o incluso el mismo broker de Adafruit permite graficar los mensajes que recibe en los diferentes topics. Aún así se pensó realizar algo más personalizado.

Para ello se ha diseñado una aplicación en Python, basada en el framework web del mismo conocido como Flask, el funcionamiento de esta aplicación es muy sencilla pues al entrar en ella nos encontraremos una página de entrada para poder entrar con un usuario y contraseña exclusivo. Al entrar podemos configurar hasta dos topics MQTT del que la aplicación estará consumiendo y pintando unas gráficas de sus evoluciones temporales. Cabe destacar que para poder acceder a dicha aplicación como una web con su url, como cualquier página en internet, se ha implementado en Heroku. Este es un servicio en la nube que nos permite tener nuestra aplicación levantada en sus servidores y acceder a ella a través de https://flask-new1.herokuapp.com/.

Para el modo sin conexión tenemos que es la propia WiPy la que nos sirve la página,es por eso que en este caso hemos tenido que optar por codificar dicha página en HTML puro junto con JavaScript.Se ha desarrollado una primera pantalla de configuración dinámica, es decir va cambiando según vayamos seleccionando configuraciones como si es Online u Offline o la cantidad de sensores a conectar. Una vez configurada nos lleva a las gráficas que andábamos buscando. Aparecerán tantas gráficas como sensores hayamos configurado y cada una recogerá los datos directamente del proceso que se esta ejecutando en WiPy a partir de autopeticiones automatizadas el endpoint ya descrito /fetchData". Aclarar que para la formación de dichas gráficas hemos usado el plugin CanvasJS, en su versión de prueba.

Las distintas páginas que hemos comentado se podrán ver con detalle en el siguiente capítulo donde hablaremos de las pruebas realizadas y describiremos imágenes de las mismas.

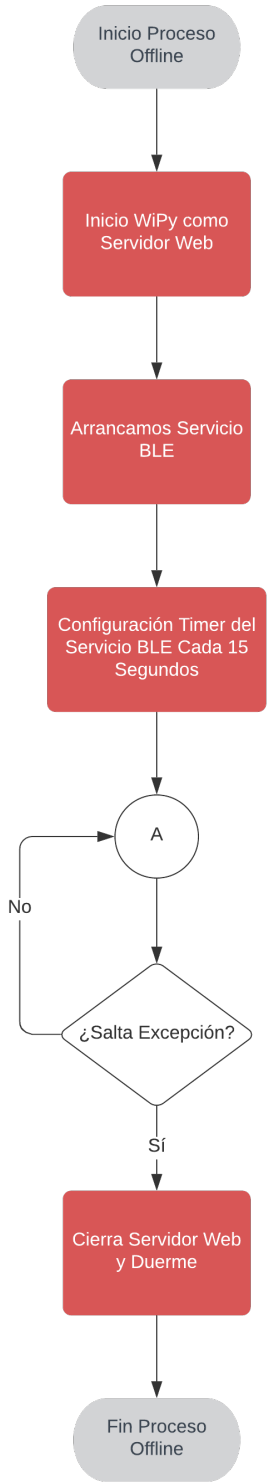

Figura 4.6: Diagrama del Proceso Offline.

Capítulo 5

Pruebas Realizadas

En este último capítulo final del desarrollo del proyecto, expondremos las pruebas de funcionamiento realizadas. Como ya sabemos tenemos dos modos de conexión disponibles así que dividiremos dichos tests en dos apartados distintos.

5.1. Pruebas en Local.

5.1.1. Prueba con un Solo Sensor.

A continuación probaremos el método de conexión offline, es decir no nos conectaremos a ninguna red WiFi externa y mostraremos todo el proceso que se ha seguido mediante las imágenes que veremos a continuación.

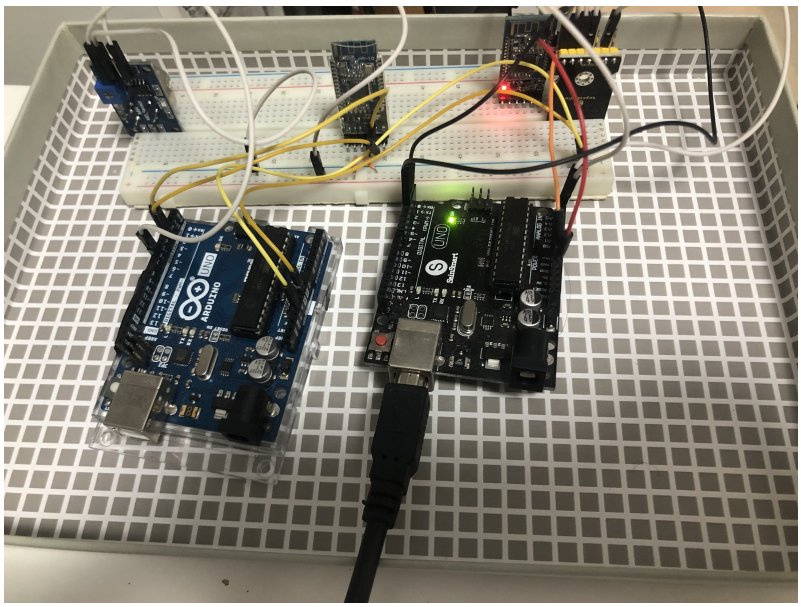

Figura 5.1: Conexión para Prueba con un Sensor.

Una vez que conectemos WiPy, vemos que está funcionará como punto de acceso a la cual nos podremos conectar, con el nombre de red "WipyConfig"

Figura 5.2: Red WiFi de Configuración.

Una vez establecemos conexión con la placa, nos vamos al navegador y accedemos a ella a través de su dirección "http://192.168.4.1/".Lo primero que veremos será la siguiente página de configuración.

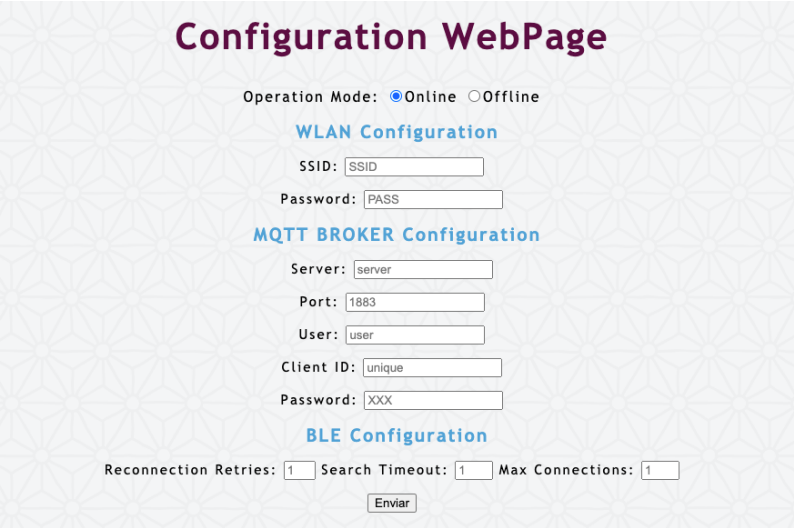

Figura 5.3: Página Web de Configuración.

Para esta prueba seleccionaremos como modo de conexión offline, y configuraremos un solo sensor BLE como podremos ver.

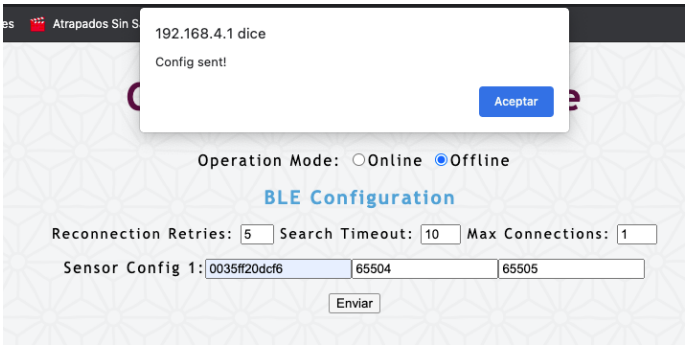

Figura 5.4: Red WiFi de Configuración.

Una vez WiPy reciba la configuración que hemos introducido, buscará dicho sensor a su alrededor y si lo encuentra intentará establecer conexión con él. En las dos siguientes imágenes de log podremos apreciar dicho proceso así como el

lanzamiento del timer, en el cual no se hace nada cada 15 segundos porue ya se encuentra conectado al sensor.

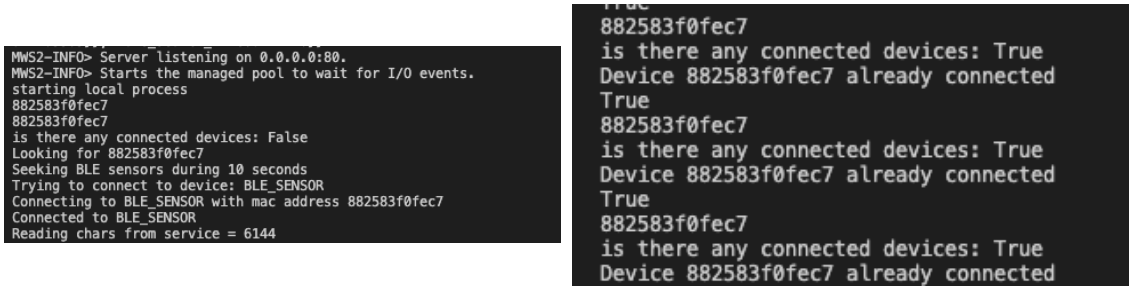

Figura 5.5: Logs Proceso Offline.

Tenemos todo configurado y el sensor conectado mandando datos, así que solo nos faltaría ver la evolución del mismo. Para ello vemos que WiPy al estar en modo offline servirá otr red WiFi, llamada "Floor1.[en] este caso, pues tendremos que conectarnos a ella y en el navegador volver a introducir la dirección de la placa, donde ya podremos ver la gráfica. Se aprecia dicha evolución como el sensor funciona perfectamente presentando dos picos intencionados.

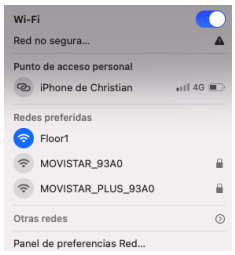

Figura 5.6: Red WiFi de Visualización.

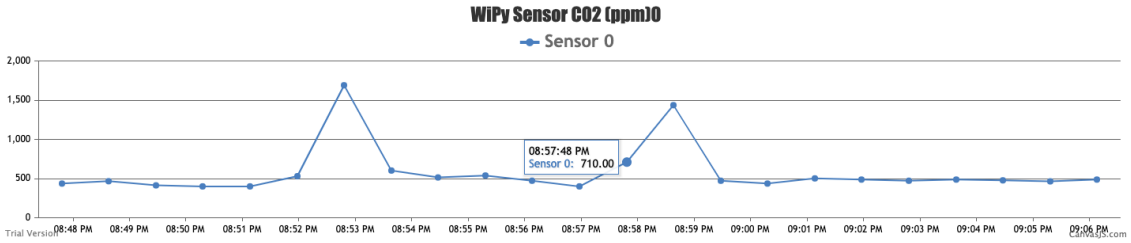

Figura 5.7: Evolución Temporal Valores de Sensor.

5.1.2. Prueba con Varios Sensores.

A continuación seguiremos el mismo proceso pero conectando dos sensores en vez de uno. En esta prueba podremos observar también que ocurre cuando uno de los sensores de la red pierde conexión. Para comenzar dicha prueba conectamos el otro módulo que tenemos en nuestra conexión tal y como vimos en la prueba anterior.

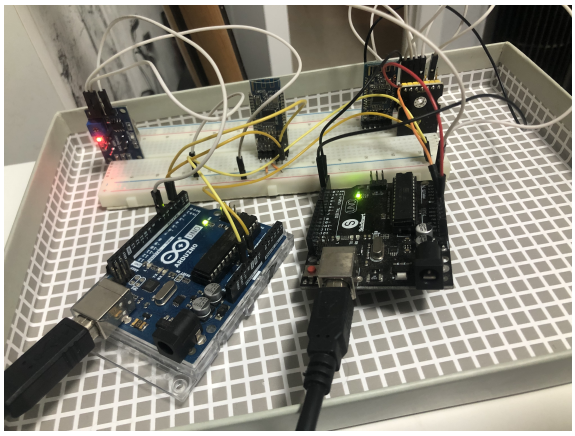

Figura 5.8: Conexión para Prueba con dos Sensores.

El proceso inicial será el mismo que el anterior, pues debemos conectarnos a la red creada "WipyConfigz configurar los dos sensores que tenemos disponibles en la misma web.

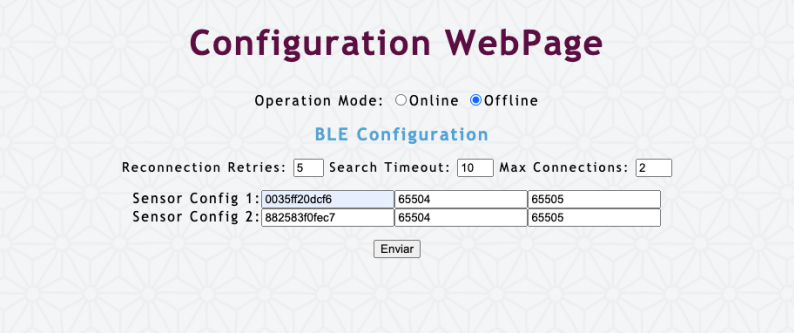

Figura 5.9: Configuración Múltiples Sensores.

Wipy recibirá la configuración de los dos sensores y tratará de conectarse a ellos, una vez establezca conexión podremos ver la visualización de dichos datos de la misma forma que anteriormente, la diferencia es que ahora podremos ver dos gráficas, una para cada sensor. Como vamos a apreciar en las gráficas de datos, tenemos el CCS811 trabajando con bastantes buenos resultados, pero el otro sensor que hemos colocado es el defectuoso MQ-135, en el cual recibimos medidas que no nos sirven, pero no contabamos con dos sensores CCS811.

Finalmente, para hacer una prueba de desconexión, lo que haremos será desconectar un módulo de su alimentación. Podemos ver que se pierde la conexión en WiPy, pero gracias a la interrupción que tenemos cada 15 segundos, detecta dicha pérdida de conexión y trata de volver a tomar dicha conexión las veces que nosotros le hayamos configurado. Podemos apreciar que durante la búsqueda volvemos a alimentar el módulo BLE, y es capaz de retomar la conexión sin ningún tipo de problema.

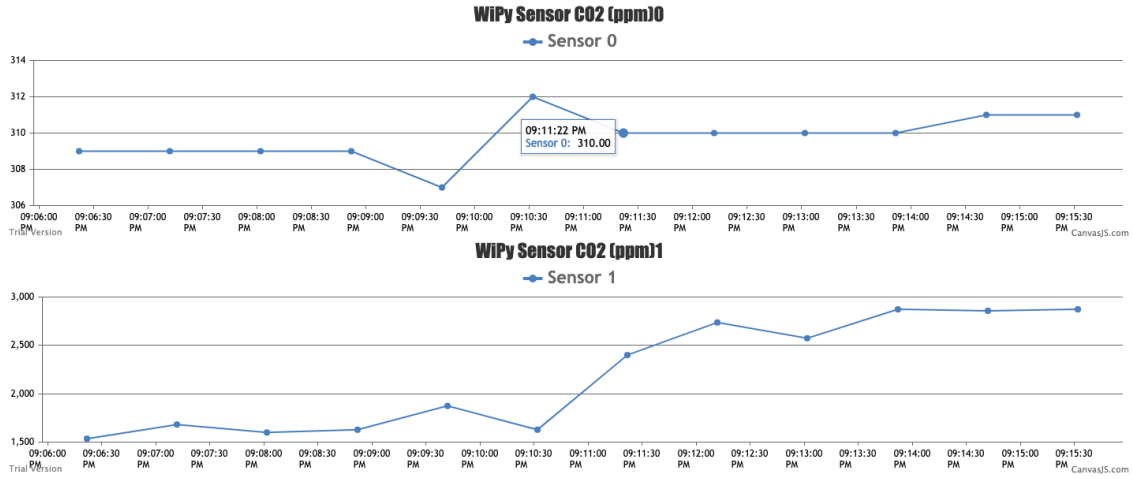

Figura 5.10: Evolución Temporal de Datos de Múltiples Sensores.

```
is there any connected devices: True
Device 0035ff20dcf6 already connected
True
882583f0fec7
is there any connected devices: True
Looking for 882583f0fec7
Seeking BLE sensors during 10 seconds
Looking for 882583f0fec7
Seeking BLE sensors during 10 seconds
```

```
Trying to connect to device: BLE_SENSOR
Connecting to BLE_SENSOR with mac address 882583f0fec7
Connected to BLE_SENSOR
Reading chars from service = 6144
char 10752 value = b'BLE_SENSOR'
char 10753 value = b'\x00\x00'
char 10754 value = b'\x00'
char 10756 value = b'P\x00\xa0\x00\x00\x00\xe8\x03'
Reading chars from service = 6145
Reading chars from service = 6154
char 10787 value = b'\xc7\xfe\xf0\x00\x00\x83%\x88'
char 10788 value = b'Model Number\x00'
char 10789 value = b'Serial Number\x00'
char 10790 value = b'Firmware Revision\x00'
char 10791 value = b'Hardware Revision\x00'
char 10792 value = b'Software Revision\x00'
char 10793 value = b'Manufacturer Name\x00'
char 10794 value = b'\xfe\x00experimental'
char 10832 value = b'\x01\r\x00\x00\x00\x10\x01'
```

Figura 5.11: Logs Reconexión.

5.2. Prueba Online.

A continuación podremos ver como se desarrolla el modo de funcionamiento online. Para comenzar su uso tenemos que como ya sabemos introducir la configuración, para ello ya sabemos que tenemos que acceder a la web de configuración con la red WiFi y la dirección de la placa,rellenando la configuración online para finalizar.

Una vez mandemos la configuración WiPy iniciará su proceso de conexión a la red que hayamos configurado y al broker MQTT.

A continuación ya podremos acceder a nuestra aplicación Flask a través de la dirección https://flask-new1.herokuapp.com/. Lo primero que nos encontraremos será una página para introducir nuestro usuario y contraseña predefinidas. Ya tenemos acceso y tendremos que indicarle a la aplicación de que dos topics vamos a consumir, no hace falta indicarle más configuración del broker MQTT porque ya esta internamente ajustado a nuestros datos. Finalmente podremos observar dos gráficas que muestran la evolución de los datos recibidos para ambos topics configurados.

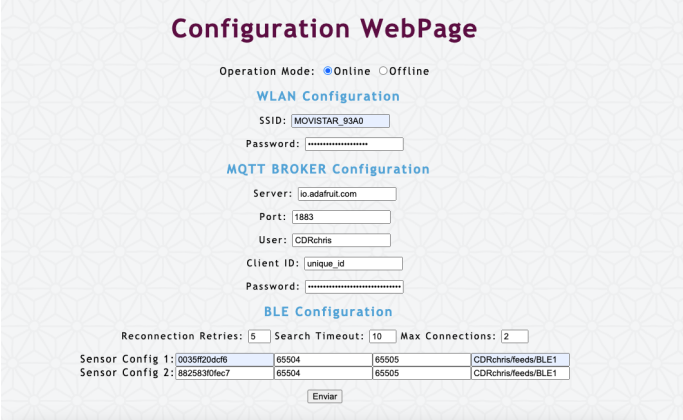

Figura 5.12: Web de Configuración Online.

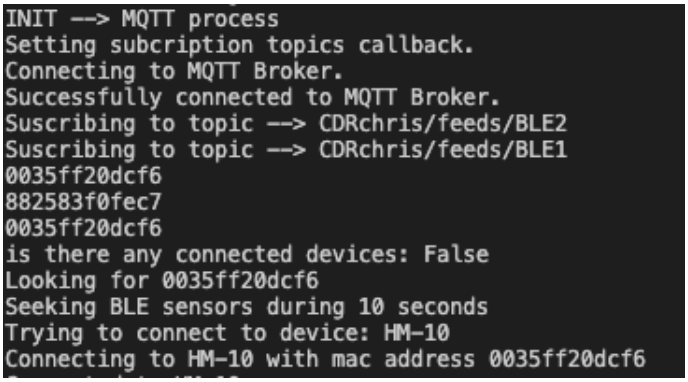

Figura 5.13: Log Inicial Proceso Online

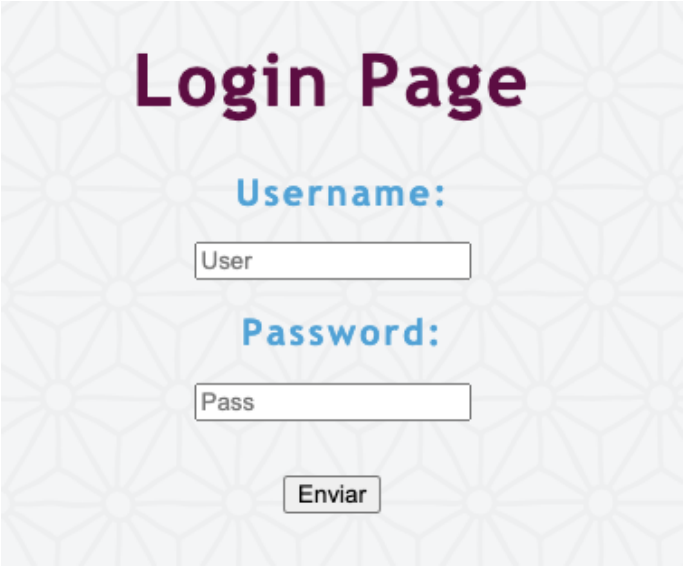

Figura 5.14: Página de Acceso Inicial.

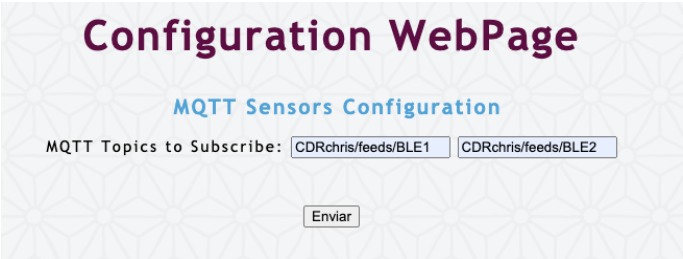

Figura 5.15: Página de Configuración de Topics.

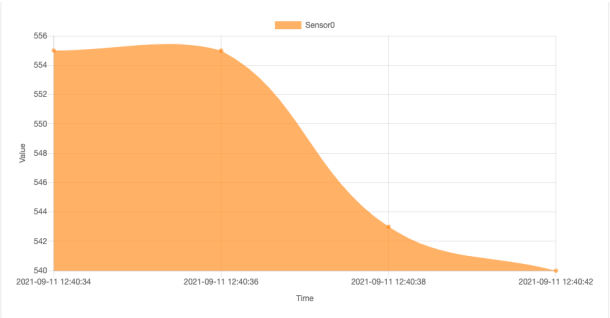

Figura 5.16: Gráfica de Datos Sensor 1.

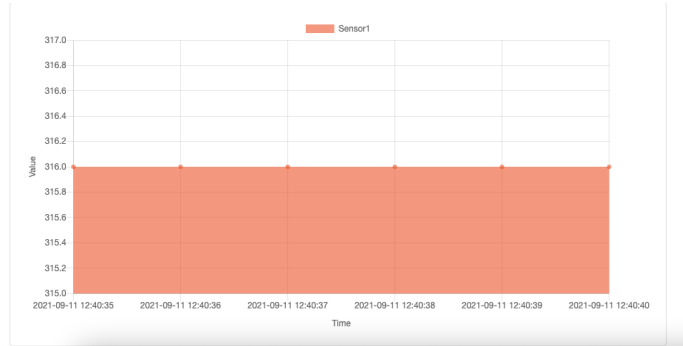

Figura 5.17: Gráfica de Datos Sensor 2.

Capítulo 6

Futuras Líneas.

En este apartado final comentaremos algunos puntos en los que podremos hacer hincapié una vez finalizado este proyecto, a fin de intentar mejorar lo ya realizado, o simplemente poder evolucionar el mismo en un futuro.

La primera pregunta más importante una vez tenemos la red de sensores BLE desarrollada sería si podríamos actuar de algún modo frente a las medidas que obtenemos. Tomemos en cuenta que si recibimos medidas de nuestros sensores de CO_2 anormales, las cuales podrían poner en peligro la ejecución de un proceso, o incluso la salud de los presentes en una estancia, no podemos hacer mas que levantar la voz de alarma. Es por eso que una evolución natural del desarrollo del proyecto sería implementar una segunda red de dispositivos, pero en este caso serían actuadores y no sensores. Con esta red de actuadores podríamos controlar las medidas que tomemos, como en el caso de la calidad del aire encender automáticamente y a distancia el mecanismo de ventilación de una habitación. Una vez esto estuviera implementado obtendríamos un proceso totalmente automatizado de control personalizado para cada una de las situaciones en las que se implementara.

Otra opción no tan atractiva en cuanto evolución, pero si que mejoraría lo ya desarrollado, sería realizar un estudio intensivo acerca del consumo de los servidores BLE instalados. Se podría mejorar el conjunto de microcontrolador-sensor-BLE, para minimizar el consumo de energía de todo este conjunto. Gracias a esto podríamos llegar a conseguir un producto final que funcionará a pilas sin preocuparnos de que se pudieran acabar, siendo totalmente inalámbrico para ser implementado en cualquier lugar que dispongamos.

Finalmente y en relación al conjunto del que mencionábamos anteriormente, podríamos sustituir el microcontrolador Arduino usado por otro mucho más básico y eficiente energéticamente hablando. Este microcontrolador no necesita apenas recursos pues su tarea es bastante sencilla, así pues la optimización de dicho proceso sería un punto a evolucionar en futuros desarrollos sobre este proyecto. A la hora de poder producir todo lo desarrollado como un producto esto traería múltiples ventajas respecto al precio pudiendo optimizar el proceso.

Capítulo 7

Anexo

7.1. Código WiPy.

7.1.1. Librería BLE.

```
from network import Bluetooth
import time
import gc
from log import log
import ubinascii

class BLE():
    def __init__(self, config,mac,client_mqtt=None,online=
        True):
        self.mac=mac
        self.sensor_config=config['sensor_config'][mac]
        self.client_mqtt=client_mqtt
        if self.client_mqtt:
            self.topic=config['sensor_config'][mac]['
                mqtt_topic']
        self.max_conn = config['max_ble_conn']
        self.ble_search_timeout = config["ble_search_timeout
            "]
        self.reconnection_retries = config["
            reconnection_retries"]
        self.connected_devices=False
        self.last_value=0
        self.bt=Bluetooth()
        self.conn=None


    @staticmethod
    def notification_callback(kwargs,info):
        char_object = kwargs[0]
        self = kwargs[1]
```

```
        self.last_value=char_object.value().decode().replace
            ("\r\n","")
        log.data("{};{}".format(self.mac,self.last_value))
        if self.client_mqtt is not None:
            try:
                self.client_mqtt.pub(topic=self.topic, msg=
                    char_object.value().decode())
            except Exception as e:
                log.info(e)

    def init_ble_search(self,mac):
        """
            First method BLE
        Args:
        Returns:
            adv:
            mac_string:
            ble_name:
        """
        try:
            log.info("Looking for {} ".format(self.mac))
            log.info("Seeking BLE sensors during {} seconds"
                .format(self.ble_search_timeout))
            self.bt.start_scan(self.ble_search_timeout)
            while self.bt.isscanning():
                adv = self.bt.get_adv()
                if adv:
                    mac_found = ubinascii.hexlify(adv.mac).
                        decode()
                    if mac_found == self.mac:
                        self.bt.stop_scan()
                        ble_name = self.bt.resolve_adv_data(
                            adv.data, Bluetooth.ADV_NAME_CMPL
                            )
                        return adv,mac_found,ble_name
                    else:
                        continue
            return None,None,None
        except Exception as e:
            log.info("EXCEPT EN INIT BLE SEARCH")
            log.info(e)
            return None,None,None

    def connect_to_ble(self,adv,ble_name):
        log.info("Connecting to {} with mac address {}".
            format(ble_name, self.mac))
```

```
        try:
            self.conn = self.bt.connect(adv.mac)
            #self.bt.callback(trigger=Bluetooth.
                CLIENT_DISCONNECTED, handler=self.
                lost_connection_callback)
        except Exception as e:
            return None
            log.info(e)

    def main_ble_process(self):
        try:
            log.info("is there any connected devices: {}".
                format(self.connected_devices))
            if self.conn is not None:
                if self.conn.isconnected():
                    log.info("Device "+self.mac+" already
                        connected")
                    return

            for i in range(0,self.reconnection_retries):
                adv,mac,name = self.init_ble_search(self.mac
                    )
                if name:
                    for i in range(0,self.
                        reconnection_retries):
                        time.sleep(2)
                        log.info("Trying to connect to
                            device: {}".format(name))
                        self.connect_to_ble(adv,name)
                        if self.conn.isconnected():
                            log.info("Connected to {}".
                                format(name))
                            self.connected_devices=True
                            break

                    services = self.conn.services()
                    for service in services:
                        log.info('Reading chars from service
                            = {}'.format(service.uuid()))
                        chars = service.characteristics()
                        for char in chars:
                            if (char.properties() &
                                Bluetooth.PROP_READ):
                                log.info('char {} value = {}
                                    '.format(char.uuid(),
                                    char.read()))
```

```
                                if service.uuid() is self.
                                   sensor_config['service_uuid']
                                    and char.uuid() is self.
                                   sensor_config['char_uuid']:
                                    s = "Server connected to
                                       Client"
                                    char.write(s.encode())
                                    char.callback(trigger=
                                       Bluetooth.
                                      CHAR_NOTIFY_EVENT,
                                       handler=self.
                                       notification_callback,
                                       arg=[char, self])
                                    return

            log.info("No se ha encontrado ningun sensor")

        except Exception as e:
            log.info(e)
            log.info("EXCEPT EN MAIN_BLE_PRCESS")
```

7.1.2. Librería MQTT.

```
import ubinascii
import machine
import pycom
from log import log
from umqtt import MQTTClient


class MQTT():
    def __init__(self, config):
        self.server = config['broker_server']
        self.port = config['broker_port']
        self.user = config['broker_user']
        self.password = config['broker_password']
        self.client_id = ubinascii.hexlify(machine.unique_id
           ())
        self.topic_list_sub = config['topic_to_suscribe']
        self.client = MQTTClient(
            self.client_id, self.server, self.port, self.
               user, self.password)

    def mqtt_configuration(self):
        self.subcription_callback()
        self.connect_client()
        self.sub()
```

```
def alive(self):
    return self.client.check_msg()

def connect_client(self):
    try:
        log.info("Connecting to MQTT Broker.")
        self.client.connect()
        log.info("Successfully connected to MQTT Broker.
            ")

    except Exception as e:
        log.info("Problem trying to connect to MQTT
            Broker.")
        log.info(e)

def disconnection(self):
    return self.client.disconnect()

def subcription_callback(self):
    try:
        log.info("Setting subcription topics callback.")
        self.client.set_callback(self.suscribe_callback)
    except Exception as e:
        log.info(e)

def sub(self):
    for topic in self.topic_list_sub:
        try:
            log.info("Suscribing to topic --> " + topic)
            self.client.subscribe(topic)
        except Exception as e:
            log.info("Unable to subscribe to topic --> "
                + topic)


def pub(self, topic, msg):
    self.client.publish(topic, msg)

@staticmethod
def suscribe_callback(topic, msg):
    log.info((topic, msg.decode()))
    string_in = msg.decode()
    try:
        float_in = float(string_in)
        log.info("Received a Number")
```

```
        except:
            log.info("Received a string")
            if msg == b"ON":
                pycom.rgbled(0xffffff)
            elif msg == b"OFF":
                pycom.rgbled(0x000000)
```

7.1.3. Librería Logging.

```
class log:

    def info(msg):
        print(msg)
        with open('/sd/log.txt', 'a') as file:
            file.write(msg+"\n")

    def data(msg):
        print(msg)
        with open('/sd/data.txt', 'a') as file:
            file.write(msg+"\n")

    def clear():
        with open('/sd/data.txt', 'w') as file:
            file.write("")
        with open('/sd/log.txt', 'w') as file:
            file.write("")
```

7.1.4. Librería Funciones Útiles.

```
import json
from log import log

import machine
from network import WLAN


def parse_config():
    with open("/flash/config.json", 'r') as conf_file:
        config =  json.loads(conf_file.read())
    return config

def write_config(dict_config):
    with open("/flash/config.json", 'w') as conf_file:
        conf_file.write(json.dumps(dict_config))


def connect_wifi(wifi_ssid, wifi_pass):
```

```
    wlan = WLAN(mode=WLAN.STA)
    wlan.connect(wifi_ssid, auth=(WLAN.WPA2, wifi_pass),
        timeout=5000)
    while not wlan.isconnected():
        machine.idle()
    log.info(wlan.ifconfig())
    return wlan


def ble_notification_callback():
    pass
```

7.1.5. Boot.py.

```
from util_functions import write_config,parse_config
import json
from MicroWebSrv2  import *
from time          import sleep
from network import WLAN
import pycom
import time
import machine
import gc
from machine import SD
from log import log

sd = SD() #create SD card object
os.mount(sd, '/sd')
log.clear()

log.info("Starting WiPy Initial WebPage.")

def connect_wlan_sta(ssid,password):
    print("connecting to new wifi")
    wlan = WLAN(mode=WLAN.STA)
    wlan.connect(ssid=ssid, auth=(WLAN.WPA2, password))
    while not wlan.isconnected():
        machine.idle()
    print("WiFi Connected")
    print(wlan.ifconfig())

pycom.heartbeat(False)
time.sleep(0.1)
pycom.rgbled(0xff0000)

global number
number=0
```

```
global start_main
start_main=False
global config
config = parse_config()
log.info(json.dumps(config))

write_config({})

@WebRoute(GET, '/LED')
def RequestTestRedirect(microWebSrv2, request) :
    global number
    number=number+1
    request.Response.ReturnOk(str(number))


@WebRoute(POST, '/config', name='ConfigPage')
def RequestTestPost(microWebSrv2, request) :
    global config
    global start_main

    data = request.GetPostedURLEncodedForm()
    print(data)
    log.info(json.dumps(config))

    try :
        config['default_mode']=data['status']

        config['ble_config']={}
        config['ble_config']['sensor_config']={}
        config['ble_config']['max_ble_conn']=int(data['conn'
            ])
        config['ble_config']['ble_search_timeout']=int(data[
            'timeout'])
        config['ble_config']['reconnection_retries']=int(
            data['retries'])

        if data['status'] =='online':
            config['wlan_config']={}
            config["wlan_config"]["wlan_ssid"] = data['SSID'
                ]
            config["wlan_config"]["wlan_password"] = data['
                PASSWORD']

            config['MQTT_config']={}
            config['MQTT_config']['broker_server']=data['
                mqtt_server']
```

```
                config['MQTT_config']['broker_port']=int(data['
                    mqtt_port'])
                config['MQTT_config']['broker_user']=data['
                    mqtt_user']
                config['MQTT_config']['broker_password']=data['
                    mqtt_pass']
                config['MQTT_config']['broker_client_id']=data['
                    client_id']
                config['MQTT_config']['topic_to_suscribe']=[]


            for i in range(0,int(data['conn'])):

                config['ble_config']['sensor_config'][data['mac'
                    +str(i)]]={}
                config['ble_config']['sensor_config'][data['mac'
                    +str(i)]]['service_uuid']=int(data['service'+
                    str(i)])
                config['ble_config']['sensor_config'][data['mac'
                    +str(i)]]['char_uuid']=int(data['char'+str(i)
                    ])

                if data['status'] == 'online':
                    config['ble_config']['sensor_config'][data['
                        mac'+str(i)]]['mqtt_topic']=data['topic'+
                        str(i)]
                    config['MQTT_config']['topic_to_suscribe'].
                        append(data['topic'+str(i)])

            write_config(config)
            request.Response.ReturnOk("OK")
            start_main=True

        except Exception as e:
            log.info(e)
            request.Response.ReturnBadRequest()
            return

wifi = WLAN()
wifi.init(mode=WLAN.AP, ssid='WipyConfig', auth=None,
    channel=2, antenna=WLAN.INT_ANT)
log.info("WiFi is up!")
log.info(str(wifi.ifconfig()))
mws2 = MicroWebSrv2()
mws2.RootPath = "/flash/www/config"
mws2.SetEmbeddedConfig()
```

```
mws2.NotFoundURL = '/'
mws2.StartManaged()
pycom.rgbled(0xffd7000)

while mws2.IsRunning and not start_main:
    sleep(1)
mws2.Stop()
gc.collect()
```

7.1.6. Main.py.

```
from network import WLAN        # For operation of WiFi
    network
import time                     # Allows use of time.sleep()
    for delays
import pycom                    # Base library for Pycom
    devices
from log import log
import json
from umqtt import MQTTClient    # For use of MQTT protocol to
    talk to Adafruit IO
import ubinascii                # Needed to run any
    MicroPython code
import machine                  # Interfaces with hardware
    components
import micropython              # Needed to run any
    MicroPython code
from util_functions import parse_config
from online import online_main
from local import local_main
from MicroWebSrv2  import *


config = parse_config()
log.info(json.dumps(config))

if config['default_mode'] == 'online':
    online_main(config)
else:
    local_main(config)
```

7.1.7. Proceso Offline.

```
import pycom
import time
from log import log
from  BLE_process import BLE
```

```
import machine
from machine import Timer
from network import Bluetooth
from MicroWebSrv2  import *
from network import WLAN

global end
end=False
global size
size=0
global listResponse
listResponse=[]
global ble_connections
ble_connections=[]

@WebRoute(GET, '/fetchData')
def RequestTestRedirect(microWebSrv2, request) :
    global listResponse
    listResponse=[]
    print(getattr(ble_connections[0],'last_value'))
    for connection in ble_connections:
        listResponse.append(int(getattr(connection,'
            last_value')))
    #for element in range(0,len(listResponse)):
    #    listResponse[element]=listResponse[element]+0.01
    request.Response.ReturnOk(str(listResponse))

@WebRoute(GET, '/graphs')
def RequestTestRedirect(microWebSrv2, request) :
    global size
    request.Response.ReturnOk(str(size))

def local_main(config):
    global end
    global ble_object
    global size
    global listResponse
    global ble_connections


    size=config['ble_config']['max_ble_conn']


    mws2_local=init_web_server()
    pycom.rgbled(0x00ff00)
    log.info("starting local process")
```

```
    pycom.heartbeat(False)
    time.sleep(0.1)
    pycom.rgbled(0xff0000)

    for mac in config['ble_config']['sensor_config'].keys():
        print(mac)
        ble_connections.append(BLE(config['ble_config'],mac,
            None,online=False))

    ble_service(config['ble_config'])
    Timer.Alarm(ble_service, 15,arg=config['ble_config'],
        periodic=True)


    pycom.rgbled(0x00ff00)
    while not end:
        time.sleep(1)
    pycom.rgbled(0xff0000)
    mws2_local.Stop()
    log.info('Going to a deepsleep')
    machine.deepsleep()




def ble_service(ble_config):
    global ble_connections

    for index,mac in enumerate(ble_config['sensor_config'].
        keys()):
        log.info(mac)
        ble_connections[index].main_ble_process()
        print(ble_connections[index].connected_devices)


def init_web_server():
    wifi = WLAN()
    wifi.init(mode=WLAN.AP, ssid='Floor1', auth=None,
        channel=2, antenna=WLAN.INT_ANT)
    mws2_local = MicroWebSrv2()
    mws2_local.RootPath = "/flash/www/local"
    mws2_local.SetEmbeddedConfig()
    mws2_local.NotFoundURL = '/'
    mws2_local.StartManaged()
    return mws2_local
```

7.1.8. Proceso Online.

```
import pycom
import time
from log import log
from  BLE_process import BLE
from machine import Timer
from mqtt_wipy import MQTT
from network import Bluetooth
from util_functions import connect_wifi

global ble_connections
ble_connections=[]

def online_main(config):
    global ble_connections
    log.info("Starting␣ONLINE␣Wipy␣process")
    pycom.heartbeat(False)
    time.sleep(0.1)
    pycom.rgbled(0xff0000)
    wlan_config = config['wlan_config']
    log.info("Connecting␣to␣Wifi,␣credentials␣from␣JSON")
    wlan = wifi_connection(wlan_config['wlan_ssid'],
       wlan_config['wlan_password'])
    client_mqtt = mqtt_process(config['MQTT_config'])

    for mac in config['ble_config']['sensor_config'].keys():
        print(mac)
        ble_connections.append(BLE(config['ble_config'],mac,
           client_mqtt,online=True))
    ble_service(config['ble_config'])
    Timer.Alarm(ble_service, 15,arg=config['ble_config'],
       periodic=True)


    try:
        pycom.rgbled(0x00ff00)
        while True:
            client_mqtt.alive()
    finally:
        client_mqtt.disconnection()
        client_mqtt = None
        wlan.disconnect()
        wlan = None
        pycom.rgbled(0x000022)
        log.info("Disconnected␣from␣Broker")
```

```
def ble_service(ble_config):
    global ble_connections

    for index,mac in enumerate(ble_config['sensor_config'].
       keys()):
        log.info(mac)
        ble_connections[index].main_ble_process()
        print(ble_connections[index].connected_devices)



def wifi_connection(ssid, password):
    try:
        wlan_object = connect_wifi(ssid, password)
        log.info("Connected to Wifi")
        pycom.rgbled(0xffd7000)
        return wlan_object
    except Exception as e:
        log.info("Problem connecting to Wifi")
        log.info(e)


def mqtt_process(config):
    try:
        log.info("INIT --> MQTT process")
        client = MQTT(config)
        client.mqtt_configuration()
        return client
    except Exception as e:
        log.info("Problem with MQTT config JSON")
        log.info(e)
        return None
```

7.1.9. JSON de Configuración.

```
{
    "default_mode": "online",
    "ble_config": {
        "max_ble_conn": 10,
        "ble_search_timeout": 5,
        "reconnection_retries": 5,
        "sensor_config": {
            "882583f0fec7": {
                "service_uuid": 65504,
                "char_uuid": 65505,
```

```
                "mqtt_topic": "CDRchris/feeds/BLE1"
            },
            "0035ff20dcf6": {
                "service_uuid": 65504,
                "char_uuid": 65505,
                "mqtt_topic": "CDRchris/feeds/BLE2"
            }
        }
    },
    "wlan_config": {
        "AP_name": "Floor1",
        "wlan_ssid": "Iphone de Christian",
        "wlan_password": "scoobydoopapa"
    },
    "MQTT_config": {
        "broker_server": "io.adafruit.com",
        "broker_port": 1883,
        "broker_user": "CDRchris",
        "broker_password": "aio_xfLv26HbmNR4AuXUkwDChZT5uGUz
           ",
        "broker_client_id": "unique_id",
        "topic_to_suscribe": [
            "CDRchris/feeds/lights",
            "CDRchris/feeds/BLE1",
            "CDRchris/feeds/BLE2"
        ]
    }
}
```

7.1.10. Código Web.

Web Configuración

```
<!DOCTYPE html>
<html>
    <head>
        <title>Config</title>
        <link rel="stylesheet" href="/CSS/style.css" />
        <link rel="icon" type="image/vnd.microsoft.icon"
           href="./assets/favicon.png" sizes="16x16">
    </head>
    <body>
   <div align="center">
    <section class="container">
        <div class="login">
            <h1>Configuration WebPage</h1>
```

```
<form action="/config" method="post">
    <div>
 <label for="mode">Operation Mode: </label>
        <input type="radio" name="status" id="
            onlineRadio" value="online" checked="
            checked" onchange="changeVisibility(this.
            value)">Online</input>
        <input type="radio" name="status" id="
            offlineRadio" value="offline" onchange="
            changeVisibility(this.value)">Offline</
            input>
    </div>
    <div id="wlanConfig">
        <h2>WLAN Configuration</h2>
        <p>
            <label for="SSID">SSID: </label>
            <input type="text" name="SSID" value=""
                placeholder="SSID">
        </p>
        <p>
            <label for="PASS">Password: </label>
            <input type="password" name="PASSWORD"
                value="" placeholder="PASS">
        </p>
    </div>
    <div id="brokerConfig">
        <h2>MQTT BROKER Configuration</h2>
        <p>
            <label for="mqtt_server">Server: </label
                >
            <input type="text" name="mqtt_server"
                value="" placeholder="server">
        </p>
        <p>
            <label for="mqtt_port">Port: </label>
            <input type="number" min="1" step="1"
                name="mqtt_port" placeholder="1883" /
                >
        </p>
        <p>
            <label for="mqtt_user">User: </label>
            <input type="text" name="mqtt_user"
                value="" placeholder="user">
        </p>
        <p>
            <label for="client_id">Client ID: </
```

```
                        label>
                    <input type="text" name="client_id"
                        value="" placeholder="unique">
                </p>
                <p>
                    <label for="PASS">Password: </label>
                    <input type="password" name="mqtt_pass"
                        value="" placeholder="XXX">
                </p>
            </div>
            <div id="BLE">
                <h2>BLE Configuration</h2>
                <p>
                <label for="retries">Reconnection Retries: <
                    /label>
                <input type="number" min="1" step="1" max="5
                    " name="retries" placeholder="1" />
                <label for="timeout">Search Timeout: </label
                    >
                <input type="number" min="1" step="1" max="
                    10" name="timeout" placeholder="1" />
                <label for="conn">Max Connections: </label>
                <input type="number" min="1" step="1" max="
                    10" onchange="addFields()" name="conn"
                    placeholder="1" />
                <div id="sensors"></div>
                </p>
            </div>
                <p class="submit">
                    <input type="submit" name="submit"
                        onclick="alert('Config sent!')">
                </p>
            </form>
        </div>

    </section>
</div>
    </body>
    <script type="text/javascript" src="./functions.js"></
        script>
</html>
```

Funciones JS Configuración

```
function changeVisibility(answer) {
    addFields();
    if (answer == "offline") {
```

```
        document.getElementById('wlanConfig').style.display = "
            none";
        document.getElementById('brokerConfig').style.display =
            "none";




    } else if (answer == "online") {
        document.getElementById('wlanConfig').style.display =
            "block";
        document.getElementById('brokerConfig').style.display
            = "block";

    }

}
function addFields(){
    // Number of inputs to create
    var number = document.getElementsByName("conn")[0].value;
    // Container <div> where dynamic content will be placed
    var sensor = document.getElementById("sensors");
    // Clear previous contents of the container
    var mode = document.getElementById("onlineRadio").checked
        ;
    while (sensor.hasChildNodes()) {
        sensor.removeChild(sensor.lastChild);
    }
    for (i=0;i<number;i++){
        // Append a node with a random text
        sensor.appendChild(document.createTextNode("Sensor␣
            Config␣" + (i+1)+":"));
        // Create an <input> element, set its type and name
            attributes
        var mac = document.createElement("input");
        mac.type = "text";
        mac.name = "mac" + i;
        mac.placeholder="MAC"
        sensor.appendChild(mac);

        var service = document.createElement("input");
        service.type = "number";
        service.name = "service" + i;
        service.placeholder="Service␣UUID"
        sensor.appendChild(service);
```

```
        var char = document.createElement("input");
        char.type = "number";
        char.name = "char" + i;
        char.placeholder="Charasteristic UUID"
        sensor.appendChild(char);
       if(mode==true){
          var topic = document.createElement("input");
          topic.type = "text";
          topic.name = "topic" + i;
          topic.placeholder="MQTT Topic"
          sensor.appendChild(topic);
       }
        // Append a line break
        sensor.appendChild(document.createElement("br"));
   }
}

var counterDiv = document.getElementById('receivedText');
function updateCounterUI(counter)
{
   counterDiv.innerHTML = counter;
}


function getData()
{
    var xhttp = new XMLHttpRequest();
    var radios = document.getElementsByName('status');

    for (var i = 0, length = radios.length; i < length; i++)
        {
      if (radios[i].checked) {
        console.log(radios[i].value);

        break;
      }
    }
    xhttp.onreadystatechange = function() {
        if (xhttp.readyState == XMLHttpRequest.DONE) {
           if (xhttp.status == 200) {
           console.log((xhttp.responseText));
            updateCounterUI(xhttp.responseText)
           }
           else {
              console.log('error', xhttp);
```

```
            }
        }
    };

    xhttp.open("GET", "LED", true);
    xhttp.send();
}
```

Estilo Web

```
html, body {
        margin: 0;
        padding: 0;
}

body {
        margin: 30px;
        background-image: url('../assets/WEB_BACKGROUND.png
            ');
        font-family: "Trebuchet MS", Verdana, sans-serif;
        letter-spacing: 2px;
}
h1 {
        color: #551f43;
        text-align: center;
        font-size: 40px;
}
h2 {
        color: #5FA3D3;
        padding-left: 12px;
        text-align: center;
        font-size: 20px;
}
p, ul, li, td {
        color: black;
        font-size: 15px;

}
a:link {
        color: green;
        text-decoration: underline;
}
a:visited {
        color: gray;
}
a:hover {
        color: red;
```

```
        text-decoration: none;
}
a:active, a:focus {
        color: red;
}
.sep {
        margin: 15px 0;
        height: 3px;
        background-color: #5FA3D3;
}
.label {
        color: #D35F8D;
        font-weight: bold;
}
```

Web Gráficas

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="utf-8"/>
<script>
var xhttp = new XMLHttpRequest();


window.onload =

function () {

var graph_number=0;
var req = new XMLHttpRequest();
req.open("GET", "graphs", false);
req.send();

if (req.status == 200) {
console.log((req.responseText));
graph_number=parseInt(req.responseText);
}
else {
    console.log('error', req);
}
var charts = [];
var dataTotal=[];
var newData=[];
var parentDiv = document.getElementById("chartsDiv");

for (i=0;i<graph_number;i++){
```

```
        var chartDiv = document.createElement("div");
        chartDiv.id = "chart"+i;
        chartDiv.style.cssText="height: 300px; width: 100%;"
            ;
        parentDiv.appendChild(chartDiv);
}
for (i=0;i<graph_number;i++){
        dataTotal[i]=[];
        newData[i]=0;
        charts[i]=new CanvasJS.Chart("chart"+i, {
        zoomEnabled: true,
        title: {
                text: "WiPy Sensor CO2 (ppm)"+i
        },
        axisX: {
                title: ""
        },
        axisY:{
                prefix: ""
        },
        toolTip: {
                shared: true
        },
        legend: {
                cursor:"pointer",
                verticalAlign: "top",
                fontSize: 22,
                fontColor: "dimGrey",
                itemclick : toggleDataSeries
        },
        data: [{
                type: "line",
                xValueType: "dateTime",
                yValueFormatString: "####.00",
                xValueFormatString: "hh:mm:ss TT",
                showInLegend: true,
                name: "Sensor "+i,
                dataPoints: dataTotal[i]
                }]
});
}


function toggleDataSeries(e) {
        if (typeof(e.dataSeries.visible) === "undefined" ||
           e.dataSeries.visible) {
```

```
                e.dataSeries.visible = false;
        }
        else {
                e.dataSeries.visible = true;
        }
        charts[i].render();
}


var time = new Date;
time.getHours();
time.getMinutes();
time.getSeconds();
time.getMilliseconds();

setInterval(function(){
        var req = new XMLHttpRequest();
        req.open("GET", "fetchData", false);
        req.send();

        if (req.status == 200) {
        console.log((req.responseText));
        newData = JSON.parse(req.responseText);
        time.setTime(time.getTime()+ 50000)
        console.log(newData);
        for(i=0;i<graph_number;i++){
                dataTotal[i].push({
                x: time.getTime(),
                y: newData[i]
        });
                charts[i].options.data.legendText = "␣Sensor
                   "+i+"␣"+ newData[i];
                charts[i].render();
        }
        }
        else {
                console.log('error', req);
        }
}, 5000);

}
</script>
</head>
<body>
<div id="chartsDiv"></div>
<div id="chartContainer" style="height:␣300px;␣width:␣100%;"
```

```
    ></div>
<script src="./canvas.js"></script>
</body>
</html>
```

Funciones JS Gráficas

```
function changeVisibility(answer) {
    addFields();
    if (answer == "offline") {

      document.getElementById('wlanConfig').style.display = "
          none";
      document.getElementById('brokerConfig').style.display =
          "none";




    } else if (answer == "online") {
       document.getElementById('wlanConfig').style.display =
          "block";
       document.getElementById('brokerConfig').style.display
          = "block";

    }

 }
 function addFields(){
    // Number of inputs to create
    var number = document.getElementsByName("conn")[0].value;
    // Container <div> where dynamic content will be placed
    var sensor = document.getElementById("sensors");
    // Clear previous contents of the container
    var mode = document.getElementById("onlineRadio").checked
       ;
    while (sensor.hasChildNodes()) {
       sensor.removeChild(sensor.lastChild);
    }
    for (i=0;i<number;i++){
        // Append a node with a random text
        sensor.appendChild(document.createTextNode("Sensor␣
           Config␣" + (i+1)+":"));
        // Create an <input> element, set its type and name
           attributes
        var mac = document.createElement("input");
        mac.type = "number";
```

```
        mac.name = "mac" + i;
        mac.placeholder="MAC"
        sensor.appendChild(mac);

        var service = document.createElement("input");
        service.type = "number";
        service.name = "service" + i;
        service.placeholder="Service UUID"
        sensor.appendChild(service);

        var char = document.createElement("input");
        char.type = "number";
        char.name = "char" + i;
        char.placeholder="Charasteristic UUID"
        sensor.appendChild(char);
      if(mode==true){
         var topic = document.createElement("input");
         topic.type = "text";
         topic.name = "topic" + i;
         topic.placeholder="MQTT Topic"
         sensor.appendChild(topic);
      }
       // Append a line break
       sensor.appendChild(document.createElement("br"));
   }
}

var counterDiv = document.getElementById('receivedText');
function updateCounterUI(counter)
{
   counterDiv.innerHTML = counter;
}


function getData()
{
    var xhttp = new XMLHttpRequest();
    var radios = document.getElementsByName('status');

    for (var i = 0, length = radios.length; i < length; i++)
        {
      if (radios[i].checked) {
        console.log(radios[i].value);

        break;
      }
```

```
    }
    xhttp.onreadystatechange = function() {
        if (xhttp.readyState == XMLHttpRequest.DONE) {
            if (xhttp.status == 200) {
            console.log((xhttp.responseText));
             updateCounterUI(xhttp.responseText)
            }
            else {
               console.log('error', xhttp);
            }
        }
    };

    xhttp.open("GET", "LED", true);
    xhttp.send();
}
```

7.2. Código Aplicación Flask.

7.2.1. Código Principal.

```
"""

A small Test application to show how to use Flask-MQTT.

"""

import eventlet
import json
from flask import Flask, redirect, url_for, request,
    render_template, flash, abort
from flask_mqtt import Mqtt
import datetime
from flask_socketio import SocketIO
from flask_bootstrap import Bootstrap

eventlet.monkey_patch()

app = Flask(__name__)
app.config['SECRET_KEY'] = b'_U?\xcc\r\x01\xb1\xf6x\xea\xac\
    xb2Gt\x00\xfc'
app.config['TEMPLATES_AUTO_RELOAD'] = True
app.config['MQTT_BROKER_URL'] = 'io.adafruit.com'
app.config['MQTT_BROKER_PORT'] = 1883
app.config['MQTT_USERNAME'] = 'CDRchris'
app.config['MQTT_PASSWORD'] = '
```

```
    aio_xfLv26HbmNR4AuXUkwDChZT5uGUz'
app.config['MQTT_KEEPALIVE'] = 5
app.config['MQTT_TLS_ENABLED'] = False
app.config['MQTT_CLEAN_SESSION'] = True
# Parameters for SSL enabled
# app.config['MQTT_BROKER_PORT'] = 8883
# app.config['MQTT_TLS_ENABLED'] = True
# app.config['MQTT_TLS_INSECURE'] = True
# app.config['MQTT_TLS_CA_CERTS'] = 'ca.crt'

mqtt = Mqtt(app)
socketio = SocketIO(app)
bootstrap = Bootstrap(app)

global configured
configured = False
global topics
topics = []


@socketio.on('disconnect')
def test_disconnect():
    global configured
    print('Client disconnected: ', request.sid)
    mqtt.unsubscribe_all()
    configured = False


@socketio.on('subscribe')
def handle_subscribe(json_str):
    data = json.loads(json_str)
    mqtt.subscribe(data['topic'])
    return redirect(url_for('charts_index'))


@app.route('/charts', methods=["GET", "POST"])
def charts_index():
    global topics
    global configured
    if request.method == "POST":
        print(request)
        for i in range(0, 2):
            topics.append(request.form.get("topic" + str(i),
                None))
            mqtt.subscribe(topics[i])
            configured = True
```

```
        print(topics)
    if not configured:
        abort(403)
    return render_template('index_charts.html')


@mqtt.on_message()
def handle_mqtt_message(client, userdata, message):
    global topics
    print(message)
    data = dict(
        topic=message.topic,
        payload=message.payload.decode()
    )
    data_point = data['payload']
    dict_data = {
        'timestamp': datetime.datetime.now().strftime('%Y-%m
          -%d %H:%M:%S'),
        'data': float(data_point)
    }
    print('received data: ' + json.dumps(dict_data))
    if data.get('topic') == topics[0]:
        socketio.emit('topicMsg0', json.dumps(dict_data))
    elif data.get('topic') == topics[1]:
        socketio.emit('topicMsg1', json.dumps(dict_data))


@mqtt.on_log()
def handle_logging(client, userdata, level, buf):
    print(level, buf)


@app.route('/', methods=["GET", "POST"])
def login_first_index():
    global configured
    if configured:
        abort(403)
    if request.method == "POST":
        username = request.form.get("username", None)
        password = request.form.get("password", None)
        if username != "CDR":
            flash('Invalid User', 'error')
            return render_template("simple_login.html")
        elif password != "1234":
            flash('Invalid Password', 'error')
            return render_template("simple_login.html")
```

```
        elif username == "CDR" and password == "1234":
            flash("Login successful")
            mqtt.unsubscribe_all()
            configured = True
            return render_template('index.html')

    elif request.method == "GET":
        return render_template("simple_login.html")


if __name__ == '__main__':
    # important: Do not use reloader because this will
       create two Flask instances.
    # Flask-MQTT only supports running with one instance
    socketio.run(app, port=5000, use_reloader=False, debug=
       False)
```

7.2.2. Web Login.

```
{% with messages = get_flashed_messages(with_categories=true
    ) %}
  {% if messages %}
    <ul class=flashes>
    {% for category, message in messages %}
      <li class="{{ category }}">{{ message }}</li>
    {% endfor %}
    </ul>
  {% endif %}
{% endwith %}
<!DOCTYPE html>
<html>
<head>
    <title>Login Page</title>
    <link rel="stylesheet" href="{{ url_for('static', 
       filename='css/style.css') }}">
    <link rel="icon" type="image/vnd.microsoft.icon" href="
       {{ url_for('static', filename='assets/favicon.png') 
       }}"
          sizes="16x16">

</head>
<body>
<div id="login">
    <div id="title" align="center">
        <h1>Login Page</h1>
    </div>
    <div id="form" align="center">
```

```
        <form method="post">
            <div>
                <h2>Username: </h2>
                <input type="text" name="username"
                    placeholder="User" required="required"/>
            </div>
            <div style="margin-bottom: 30px;">
                <h2>Password: </h2>
                <input type="password" name="password"
                    placeholder="Pass" required="required"/>
            </div>
            <div>
                <input type="submit" name="submit">
            </div>
        </form>
    </div>

</div>
</body>
</html>
```

7.2.3. Web Principal.

```
<!DOCTYPE html>
<html>
<head>
    <title>MQTT Subscription</title>
    <link rel="stylesheet" href="{{ url_for('static', 
        filename='css/style.css') }}">
    <link rel="icon" type="image/vnd.microsoft.icon" href="
        {{ url_for('static', filename='assets/favicon.png') 
        }}"
            sizes="16x16">
</head>
<body>
<div align="center">
    <section class="container">
        <div class="login">
            <h1>Configuration WebPage</h1>
            <form action="/charts" method="post">
                <div id="BLE">
                    <h2>MQTT Sensors Configuration</h2>
                    <p>
                        <label for="conn">MQTT Topics to
                            Subscribe: </label>
                        <input type="text" name="topic0"
                            placeholder="MQTT Topic" style="
```

```
                                margin-bottom: 30px;"/>
                            <input type="text" name="topic1"
                                placeholder="MQTT Topic" style="
                                margin-bottom: 30px;"/>
                        </p>
                    </div>
                    <p class="submit">
                        <input type="submit" name="submit"
                            onclick="alert('Config sent!')">
                    </p>
                </form>
            </div>

    </section>
</div>
</body>
</html>
```

7.2.4. Gráficas Sensores.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Real-Time Charts with Flask</title>
<script type="text/javascript" src="//cdnjs.cloudflare.com/
   ajax/libs/socket.io/4.0.1/socket.io.min.js"></script>
    <script type="text/javascript" src="https://www.gstatic.
       com/charts/loader.js"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/
       jquery/3.4.0/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/
       Chart.js/2.8.0/Chart.min.js"></script>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
       initial-scale=1, shrink-to-fit=no">


    <link rel="stylesheet" href="https://stackpath.
       bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.
       css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1
       fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T" crossorigin="
       anonymous">

    <link rel="stylesheet" type="text/css" href="{{ url_for
       ('static', filename='main.css') }}">
```

```
</head>

<body>
    <div class="container">
        <div class="row">
            <div class="col-10">
                <div class="card">
                    <div class="card-body">
                        <canvas id="sensor0Chart"></canvas>
                    </div>
                </div>
            </div>
        </div>
        <div class="row">
            <div class="col-10">
                <div class="card">
                    <div class="card-body">
                        <canvas id="sensor1Chart"></canvas>
                    </div>
                </div>
            </div>
        </div>
    </div>

<script type="text/javascript" charset="utf-8">
var socket = io();
google.charts.load('current', {'packages':['gauge']});
google.charts.setOnLoadCallback(drawChart);

function drawChart() {
    var chartOptions = {
            responsive: true,
            tooltips: {
                mode: 'index',
                intersect: false,
            },
            hover: {
                mode: 'nearest',
                intersect: true
            },
            scales: {
                xAxes: [{
                    display: true,
                    scaleLabel: {
                        display: true,
                        labelString: 'Time'
```

```
                }
            }],
            yAxes: [{
                display: true,
                scaleLabel: {
                    display: true,
                    labelString: 'Value'
                }
            }]
        }
    };

var sensor0Data = {
  type : 'line',
  data : {
    labels: [],
    datasets : [{
      label : 'Sensor0',
      backgroundColor : 'rgba(255, 136, 0, 0.60)',
      borderColor : 'rgba(0, 0, 0, 0)',
      data : []
    }]
  },
  options : chartOptions
};
var sensor1Data = {
  type : 'line',
  data : {
    labels: [],
    datasets : [{
      label : 'Sensor1',
      backgroundColor : 'rgba(220, 100, 60, 0.60)',
      borderColor : 'rgba(0, 0, 0, 0)',
      data : []
    }]
  },
  options : chartOptions
};

const sensor0Chart = new Chart($('#sensor0Chart'),
   sensor0Data);
const sensor1Chart = new Chart($('#sensor1Chart'),
   sensor1Data);

socket.on('topicMsg0', function(data) {
    const json_data = JSON.parse(data);
```

```
        if (sensor0Data.data.labels.length === 20) {
            sensor0Data.data.labels.shift();
            sensor0Data.data.datasets[0].data.shift();
        }
        sensor0Data.data.labels.push(json_data.timestamp);
        sensor0Data.data.datasets[0].data.push(json_data.
           data);
        sensor0Chart.update();
    });
     socket.on('topicMsg1', function(data) {
        const json_data = JSON.parse(data);
        if (sensor1Data.data.labels.length === 20) {
            sensor1Data.data.labels.shift();
            sensor1Data.data.datasets[0].data.shift();
        }
        sensor1Data.data.labels.push(json_data.timestamp);
        sensor1Data.data.datasets[0].data.push(json_data.
           data);
        sensor1Chart.update();


    });
}
</script>




</body>
</html>
```

7.3. Código Arduino.

7.3.1. Código CCS811.

```
#include <CCS811.h>
#include <SoftwareSerial.h>
SoftwareSerial HM10(2, 3); // RX = 2, TX = 3
char appData;
String inData = "";
long randnum;
String string_rand = "";

CCS811 sensor;

void setup(void)
```

```
{
  Serial.begin(9600);
  Serial.println("HM10 serial started at 9600");
  HM10.begin(9600); // set HM10 serial at 9600 baud rate
  pinMode(13, OUTPUT); // onboard LED
  digitalWrite(13, LOW); // switch OFF LED

    while(sensor.begin() != 0){
        Serial.println("failed to init chip, please check if
            the chip connection is fine");
        delay(1000);
    }
    sensor.setMeasCycle(sensor.eCycle_250ms);
}
void loop() {
  delay(1000);
    if(sensor.checkDataReady() == true){

        HM10.listen();
        while (HM10.available() > 0) {   // if HM10 sends
           something then read
          appData = HM10.read();
          inData = String(appData);  // save the data in
             string format
          Serial.write(appData);
        }

        if (Serial.available()) {           // Read user
           input if available.
          HM10.write(Serial.read());
        }

        Serial.println(String(sensor.getCO2PPM()));
        HM10.println(String(sensor.getCO2PPM()));
    } else {
        Serial.println("Data is not ready!");
    }

    sensor.writeBaseLine(0x847B);
    delay(1000);
}
```

7.3.2. Código MQ-135.

```
#include <SoftwareSerial.h>
SoftwareSerial HM10(2, 3); // RX = 2, TX = 3
```

```
char appData;
String inData = "";
long randnum;
String string_rand = "";
int sensorPin = A6;
int val = 0;

void setup()
{
  Serial.begin(9600);
  Serial.println("HM10 serial started at 9600");
  HM10.begin(9600); // set HM10 serial at 9600 baud rate
  pinMode(13, OUTPUT); // onboard LED
  digitalWrite(13, LOW); // switch OFF LED

}

void loop()
{
  val = analogRead(sensorPin);
  HM10.listen();  // listen the HM10 port
  while (HM10.available() > 0) {   // if HM10 sends
     something then read
    appData = HM10.read();
    inData = String(appData);  // save the data in string
       format
    Serial.write(appData);
  }

  if (Serial.available()) {           // Read user input if
     available.
    HM10.write(Serial.read());
  }

    delay(1000);

}
```

Bibliografía

[1] KEYESTUDIO, *KS0457 keyestudio CCS811 Carbon Dioxide Air Quality Sensor*. Recuperado de https://wiki.keyestudio.com/KS0457_keyestudio_CCS811_Carbon_Dioxide_Air_Quality_Sensor.

[2] HUAMAO, *HM-10 Dataseet 13-Febrero-2020*. Recuperado de http://www.jnhuamao.cn/bluetooth40_en.zip.

[3] HANWEI SENSOR, *Technical Data MQ-135 Gas Sensor.*. Recuperado de https://www.electronicoscaldas.com/datasheet/MQ-135_Hanwei.pdf.

[4] PYCOM, *Pycom Specsheets WiPy3.0*. Recuperado de https://docs.pycom.io/gitbook/assets/specsheets/Pycom_002_Specsheets_WiPy3.0_v2.pdf.

[5] PYCOM, *Tutoriales y Ejemlos Pycom*. Recuperado de https://docs.pycom.io/tutorials/.

[6] IZERTIS 24-MAYO-2017, *Bluetooth BLE: el conocido desconocid*. Recuperado de https://ahorasomos.izertis.com/solidgear/bluetooth-ble-el-conocido-desconocido/

[7] MOZILLA, *Primeros Pasos con CanvasJS*. Recuperado de https://developer.mozilla.org/es/docs/Web/API/Canvas_API/Tutorial/Basic_usage .

[8] MICROPYTHON, *Documentación MicroPython*. Recuperado de http://docs.micropython.org/en/latest/ .

[9] ESPLORADORES, *Servidor WEB con microWebSrv2*. Recuperado de https://www.esploradores.com/microwebsrv2/.

[10] FLASK-MQTT, *Documentación Flask-MQTT*. Recuperado de https://flask-mqtt.readthedocs.io/en/latest/ .

[11] HEROKU, *Despliegue Con Git*. Recuperado de https://devcenter.heroku.com/articles/git .

[12] ADAFRUIT, *Introducción a BLE: GAP*. Recuperado de https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap